POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Babij Aleksander - s25347
Biliński Jakub - s22705
Gawrych Jarosław - s25361
Gierczuk Michał - s24333
Łasak Szymon - s26496
Mrozek Maciej - s24921
Niedźwiadek Kacper - s24662
Siwek Magdalena - s24937
Staśkiewicz Miriam - s26229
Świder Adrian - s20710
Tober Michał - s25513
Wiśniewski Michał - s24845
Yermolaiev Oleksii - s25653

Platforma do zarządzania siecią restauracji „Reservant”

Praca inżynierska napisana pod kierunkiem:

dr. inż. Mariusza Trzaski

Warszawa, luty, 2024

Streszczenie

Celem niniejszej pracy inżynierskiej było stworzenie zintegrowanego systemu wspierającego organizację i funkcjonowanie obiektu gastronomicznego. Zadaniem aplikacji jest usprawnienie i uproszczenie procesu realizacji zamówień zarówno po stronie restauracji, jak i jej klientów. Oprócz podstawowych funkcji związanych z zarządzaniem rezerwacjami i zamówieniami, system Reservant został wzbogacony o elementy społecznościowe, takie jak możliwość dodawania znajomych, organizowanie wydarzeń oraz korzystanie z czatu tekstowego. Ponadto aplikacja optymalizuje obsługę zgłoszeń dzięki integracji z biurem obsługi klienta. Wszystkie te udogodnienia mają na celu zwiększenie atrakcyjności systemu i przyciągnięcie jak największej liczby użytkowników.

System Reservant składa się z trzech głównych komponentów:

- Serwer - został zaimplementowany w języku C# z użyciem nowoczesnego frameworka ASP.NET Core,
- Aplikacja mobilna - dedykowana platformie Android, została napisana w języku Kotlin,
- Aplikacja webowa - opiera się na bibliotekach React oraz języku TypeScript.

Każdy z elementów systemu Reservant został starannie zaprojektowany, aby wspólnie tworzyć spójną, wydajną i funkcjonalną całość.

Słowa kluczowe:

Restauracja, rezerwacja, zamówienia, wydarzenia, opinie, zarządzanie lokalem gastronomicznym, strona internetowa, aplikacja mobilna, zarządzanie pracownikami, zarządzanie stanem magazynu, biuro obsługi klienta, Java, C#, Python, JavaScript, TypeScript, React, ASP.NET, Sql, Android, AndroidStudio, Kotlin, IntelliJ, Visual Studio Code.

Spis treści

1 Wstęp

Współczesny rynek gastronomiczny, w dobie intensywnego rozwoju technologii, nieustannie poszukuje innowacyjnych rozwiązań, które mogłyby poprawić jakość usług oraz zwiększyć satysfakcję klientów. Coraz większa liczba restauracji oraz innych lokali gastronomicznych wdraża nowoczesne narzędzia informatyczne, aby sprostać rosnącym oczekiwaniom klientów oraz zwiększyć efektywność zarządzania operacjami wewnętrznymi. Jednym z takich narzędzi jest Reservant.

System Reservant przede wszystkim umożliwia klientom rezerwację stolików oraz składanie zamówień w lokalach gastronomicznych. Ponadto, wspomaga właścicieli w zarządzaniu restauracją, co przyczynia się do usprawnienia codziennych operacji i zwiększenia zadowolenia zarówno klientów, jak i personelu. Dzięki prostemu i intuicyjnemu interfejsowi, aplikacja oferuje użytkownikom wygodne rozwiązania.

1.1 Cel pracy

Celem niniejszej pracy inżynierskiej było opracowanie i wdrożenie aplikacji webowej oraz mobilnej która umożliwia m.in:

- Klientom;
    - Rezerwację stolików w lokalach gastronomicznych,
    - Składanie zamówień na potrawy i napoje w sposób szybki i intuicyjny,
    - Uzyskanie informacji o dostępności stolików, aktualnych ofertach, promocjach i wydarzeniach organizowanych przez dane lokale,
    - Możliwość dodania oceny dla poszczególnych lokali lub zobaczenia istniejących opinii,
- Właścicielowi lokalu;
    - Efektywne zarządzanie rezerwacjami stolików,
    - Optymalizację procesów związanych z przyjmowaniem i realizacją zamówień,
    - Monitorowanie i analizowanie danych dotyczących rezerwacji i zamówień w celu poprawy jakości usług,
    - Usprawnienie dostaw i monitorowania produktów dostępnych w lokalu.

Praca skupia się na dostarczeniu kompleksowego rozwiązania, które integruje potrzeby zarówno użytkowników jak i właścicieli lokali, zapewniając im wygodę oraz efektywność w korzystaniu z aplikacji.

1.2 Organizacja pracy

Niniejsza praca prowadzi czytelnika przez wszystkie etapy jej tworzenia. Rozdział drugi omawia kontekst prawny, zawierający przegląd przepisów dotyczących ochrony danych osobowych oraz regulacji branżowych, w tym zgodności z RODO. Rozdział trzeci analizuje istniejące aplikacje gastronomiczne, identyfikując ich mocne i słabe strony oraz porównuje je z aplikacją Reservant.

Kolejny rozdział opisuje propozycję systemu, czyli między innymi architekturę, główne funkcje oraz scenariusze użytkowania aplikacji, definiując wymagania funkcjonalne i niefunkcjonalne. Następnie, przedstawiono wybór technologii i narzędzi wykorzystanych do realizacji projektu.

Implementacja systemu zawiera szczegółowy opis procesu kodowania, integracji modułów oraz realizacji kluczowych funkcji, wraz z fragmentami kodu i omówieniem rozwiązań technicznych.

Rozdział dotyczący interfejsu użytkownika koncentruje się na zasadach UX/UI oraz prezentuje przykłady ekranów aplikacji, kładąc nacisk na intuicyjność i użyteczność.

Rozdział ósmy skupia się na testowaniu aplikacji, opisując metody i narzędzia użyte do testów, oraz przedstawiając wyniki, które potwierdzają gotowość aplikacji do wdrożenia. Ostatnie cztery rozdziały to kolejno bibliografia oraz spis: rysunków, tabel i listingów.

1.3 Zakres pracy

Praca nad projektem została podzielona na dziewięć zespołów, w których skład zaangażowane zostało 14 osób. Wyszczególnione zostały następujące zespoły:

1 Analizy,

2 Projektowania,

3 Designu GUI,

4 Implementacji – Frontend,

5 Implementacji – Backend,

6 Implementacji – Aplikacji mobilnej,

7 DevOps,

8 Testowania,

9 Dokumentacji.

Praca była stale monitorowana przez kierownika projektu, poprzez cotygodniowe spotkania mające na celu prezentacje dokonanych postępów lub zmian. W skład poszczególnych zespołów wchodziło od dwóch do nawet ośmiu członków. Komunikacja zarówno w zespole jak i poza zespołami dokonywana była przy użyciu serwera Discord. Zadania dla członków projektu przypisywane były przez liderów poszczególnych grup w aplikacji Trello. Każde zadanie przechodziło poprzez następujące fazy:

- *To do,*
- *Doing,*
- *In review,*
- *Done.*

Dzięki tej procedurze każdy lider zespołu mógł w efektywny i łatwy sposób kontrolować postępy prac na projektem.

1.4 Struktura organizacyjna zespołu

Fragment ten prezentuje zestawienie tabel ilustrujących podział członków poszczególnych zespołów wraz z ich krótkim opisem.

Tabela 1 Kierownik zespołu. Źródło: Opracowanie własne

Kierownik Projektu Reservant
Maciej Mrozek

Tabela 2 Podział zespołu analizy. Źródło: Opracowanie własne

Zespół Analityczny	
Kierownik	Jarosław Gawrych
Członkowie Zespołu	Aleksander Babij Jakub Biliński Michał Gierczuk Szymon Łasak Magdalena Siwek Miriam Staśkiewicz Michał Tober
Zadania Zespołu	Zespół analityczny miał za zadanie przeanalizowanie problematyki zarządzania lokalami gastronomicznymi, identyfikację uczestników procesu oraz określenie ich potrzeb. W wyniku tych działań sformułowano wymagania biznesowe. Następnie zespół opracował wymagania funkcjonalne i niefunkcjonalne, stworzył scenariusze przypadków użycia oraz przygotował różnorodne diagramy, w tym model pojęciowy, model danych, diagram przypadków użycia, diagram stanów oraz diagram aktywności.

Tabela 3 Podział zespołu projektowania. Źródło: Opracowanie własne

Zespół Projektowania	
Kierownik	Jarosław Gawrych
Członkowie Zespołu	Michał Gierczuk Adrian Świder Oleksii Yermolaiev
Zadania Zespołu	Zespół projektowy odpowiadał za przekształcenie wyników pracy zespołu analitycznego w konkretną architekturę systemu. Opracowywał diagramy architektoniczne i implementacyjne, dbając o to, aby każdy element systemu był zaprojektowany z myślą o optymalnej integracji i wydajności. Jego zadaniem było znalezienie najefektywniejszych rozwiązań technicznych, które spełnią potrzeby zidentyfikowane przez zespół analityczny, przy jednoczesnym zapewnieniu spójności technicznej całego projektu.

Tabela 4 Podział zespołu GUI. Źródło: Opracowanie własne

Zespół Design GUI	
Kierownik	Miriam Staśkiewicz
Członkowie Zespołu	Jakub Biliński Jarosław Gawrych Szymon Łasak Adrian Świder Michał Tober
Zadania Zespołu	Zespół odpowiedzialny za projektowanie graficznego interfejsu użytkownika (GUI) miał za zadanie opracowanie makiet elektronicznych, które posłużyły jako wzór dla wyglądu ekranów w aplikacjach webowej i mobilnej. Ponadto zespół stworzył standardy wizualne, w tym dobór kolorystyki, zapewniając spójność i estetykę interfejsu.

Tabela 5 Podział zespołu frontendu. Źródło: Opracowanie własne

Zespół Implementacji Frontendu	
Kierownik	Jakub Biliński
Członkowie Zespołu	Jarosław Gawrych Szymon Łasak Miriam Staśkiewicz
Zadania Zespołu	Zespół frontendowy był odpowiedzialny za tworzenie interfejsów użytkownika, wykorzystując nowoczesne technologie webowe. Ich celem było zaprojektowanie i wdrożenie responsywnej i interaktywnej aplikacji. Deweloperzy frontendowi dbali o jak najbardziej spójny wygląd różnych komponentów interfejsu, zapewniając jednolitą estetykę i intuicyjność aplikacji.

Tabela 6 Podział zespołu serwera. Źródło: Opracowanie własne

Zespół Implementacji Serwera	
Kierownik	Oleksii Yermolaiev

Członkowie Zespołu	Aleksander Babij Michał Gieczuk Kacper Niedźwiadek Michał Tober Michał Wiśniewski
Zadania Zespołu	Zespół od backendu koncentrował się na opracowywaniu logiki biznesowej systemu, zapewniając jej prawidłowe działanie zgodnie z wymaganiami. Odpowiadali za projektowanie i implementację bazy danych, dbając o jej strukturę, optymalizację oraz integralność danych. Ponadto, zajmowali się tworzeniem bezpiecznej i wydajnej komunikacji między frontendem a backendem, wykorzystując odpowiednie technologie i API, które umożliwiały płynne i bezproblemowe przesyłanie danych oraz realizację zapytań użytkowników.

Tabela 7 Podział zespołu mobilki. Źródło: Opracowanie własne

Zespół Implementacji Mobilki	
Kierownik	Aleksander Babij
Członkowie Zespołu	Maciej Mrozek Kacper Niedźwiadek Michał Wiśniewski
Zadania Zespołu	Zespół odpowiedzialny za aplikację mobilną skupiał się na tworzeniu i implementacji rozwiązań dedykowanych dla systemu Android, zapewniając, że aplikacja będzie funkcjonalna, intuicyjna i łatwa w obsłudze. Dbał o to, by interfejs użytkownika był dopasowany do standardów Androida.

Tabela 8 Podział zespołu DevOps. Źródło: Opracowanie własne

Zespół DevOps	
Kierownik	Maciej Mrozek
Członkowie Zespołu	Oleksii Yermolaiev
Zadania Zespołu	Zespół DevOps był odpowiedzialny za automatyzację procesów związanych z cyklem życia oprogramowania, w tym za wdrażanie aplikacji w trybie ciągłym. Zajmował się także zarządzaniem infrastrukturą, optymalizując jej konfigurację i

	skalowalność, oraz monitorowaniem działania systemu w środowisku produkcyjnym, aby zapewnić jego stabilność i wydajność.

Tabela 9 Podział zespołu testów. Źródło: Opracowanie własne

Zespół Testów	
Kierownik	Kacper Niedźwiadek
Członkowie Zespołu	Michał Gierczuk Szymon Łasak Magdalena Siwek Adrian Świder Michał Wiśniewski
Zadania Zespołu	Zespół odpowiedzialny za testy dbał o wysoką jakość produktu, wykonując testy manualne i automatyczne, aby sprawdzić poprawność działania poszczególnych komponentów systemu. Ich zadaniem było wykrywanie błędów i niezgodności, co pozwalało na utrzymanie wysokiego standardu funkcjonalności i stabilności aplikacji.

Tabela 10 Podział zespołu dokumentacji. Źródło: Opracowanie własne

Zespół Dokumentacji	
Kierownik	Maciej Mrozek
Członkowie Zespołu	Magdalena Siwek Miriam Staśkiewicz Adrian Świder
Zadania Zespołu	Zespół odpowiedzialny za dokumentację projektową miał za zadanie opracowanie pełnej i czytelnej dokumentacji, która precyzyjnie odzwierciedlałaby przebieg i kluczowe aspekty projektu. Zespół współpracował z innymi grupami, aby upewnić się, że wszystkie istotne informacje zostały poprawnie zebrane i przedstawione, zapewniając jasność oraz zgodność z rzeczywistym stanem projektu.

1.5 Kontekst prawny

W kontekście prawnej analizy aplikacji mobilnych jak i przeglądarkowych kluczowym aspektem jest zrozumienie i przestrzeganie obowiązujących przepisów prawa dotyczących dostępności, praw autorskich i zgodności z przepisami dotyczącymi płatności online.

Aplikacja dostosowana jest do wszelkich niezbędnych przepisów dotyczących dostępności. Twórcy aplikacji przestrzegają praw autorskich do używanych materiałów, takich jak kod źródłowy, grafiki i treści, oraz posiadają odpowiednie licencje na korzystanie z narzędzi i bibliotek.

W kontekście płatności online aplikacja zapewnia zgodność z przepisami dotyczącymi bezpieczeństwa transakcji oraz ochrony danych finansowych użytkowników.

1.5.1 Ochrona danych osobowych

Ochrona danych osobowych stanowi kluczowy aspekt w kontekście aplikacji mobilnych lub przeglądarkowych, przeważnie zbierają i przetwarzają one dane osobowe użytkowników. W celu zapewnienia ochrony danych osobowych, aplikacja posiada jasną politykę prywatności, która opisuje rodzaje zbieranych danych, sposób ich przechowywania i przetwarzania, oraz cele, dla których są wykorzystywane.

Użytkownicy są informowani o zbieraniu i przetwarzaniu ich danych osobowych, oraz muszą wyrazić na to swoją zgodę. Aplikacje stosują odpowiednie środki bezpieczeństwa, takie jak szyfrowanie danych, aby chronić dane osobowe użytkowników przed nieuprawnionym dostępem, kradzieżą lub wyciekiem danych.

1.5.2 Codzienne funkcjonowanie

Codzienne funkcjonowanie aplikacji mobilnej obejmuje szereg czynności związanych z zarządzaniem, utrzymaniem i dostosowywaniem aplikacji do potrzeb użytkowników. W ramach codziennego funkcjonowania aplikacji mobilnej, twórcy zapewniają odpowiednie wsparcie techniczne dla użytkowników, aktualizowanie aplikacji w celu poprawy funkcjonalności i usuwania błędów, monitorowanie i analizę danych użytkowników w celu doskonalenia aplikacji, oraz zapewnienie zgodności z aktualnie obowiązującymi przepisami prawa.

2 Istniejące systemy zarządzania lokalami gastronomicznymi

W poniższych podrozdziałach można znaleźć przeróżne aplikacje, jak i oprogramowania gastronomiczne wspierające proces funkcjonowania różnych typów lokali. Odbiorcami wspomnianych aplikacji są potencjalni klienci poszczególnych restauracji. Zanim zespół implementacyjny rozpoczął działanie nad aplikacją, przeprowadzono wnikliwą analizę dostępnych już rozwiązań na rynku. Badanie miało na celu znalezienie silnych oraz słabych stron poszczególnych systemów gastronomicznych jak i zapoznanie się z ich funkcjonalnościami. Wyniki pracy przedstawione zostały w poniższych podrozdziałach.

2.1 POSBistro

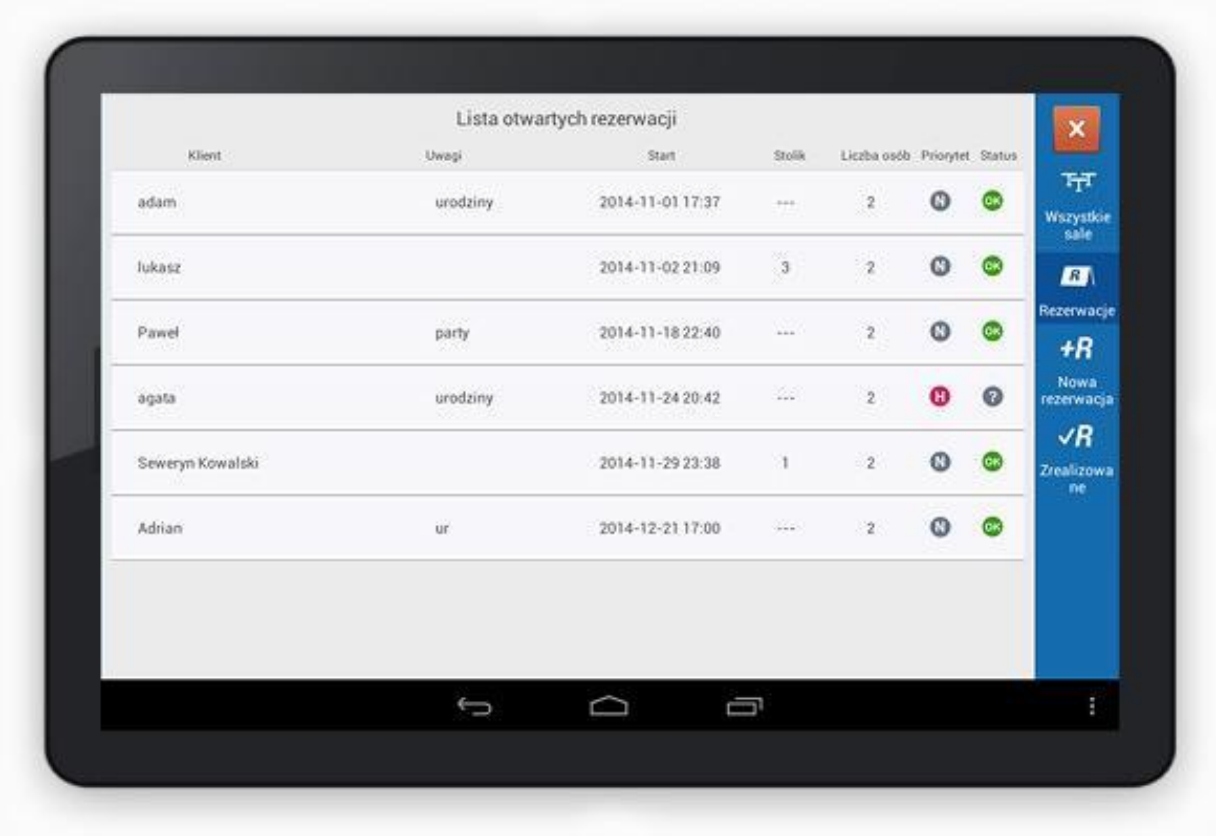
POSBistro zgodnie z [1] to polski system POS (ang. *Point of Sale*) stworzony z myślą o branży gastronomicznej który oferuje szeroki zakres funkcji wspomagających zarządzanie restauracjami, kawiarniami, barami oraz innymi obiektami gastronomicznymi. Sposób zarządzania systemem POS przedstawia Rysunek 1.

Rysunek 1 Przykład głównego ekranu POSBistro. Źródło: [1]

2.1.1 Zarządzanie rezerwacjami

Zarządzanie rezerwacjami w systemie POSBistro ułatwia organizację i koordynację w obszarze rezerwacji stolików oraz usług cateringowych w obiektach gastronomicznych. System umożliwia personelowi wprowadzanie nowych rezerwacji.

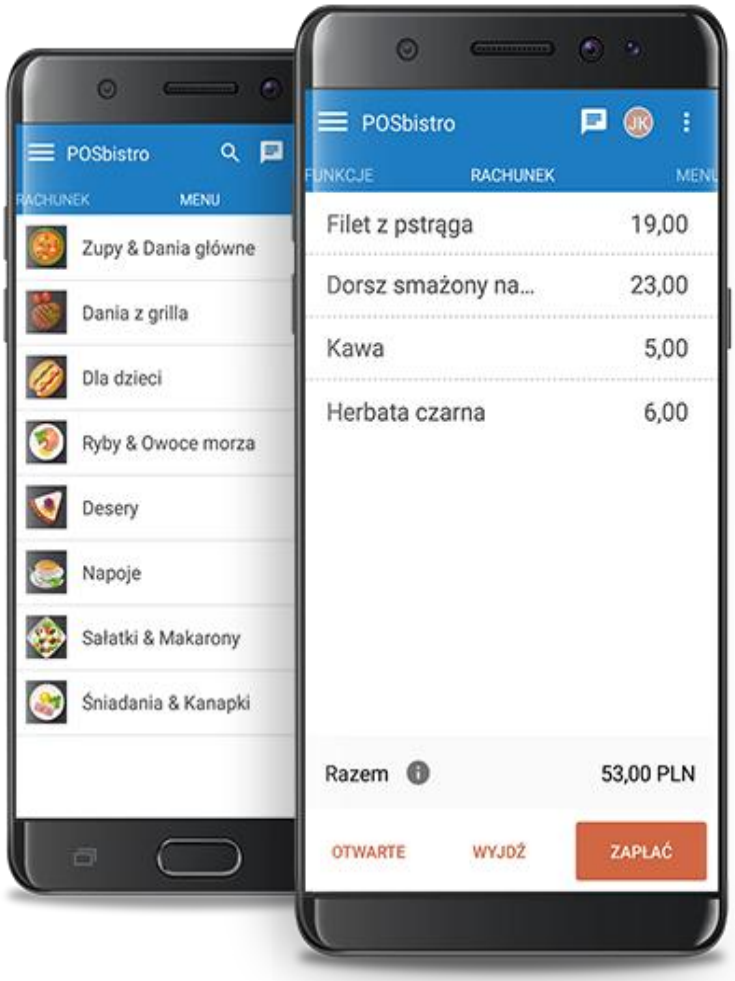
Przykład graficznego interfejsu, który pozwala na określenie liczby osób, daty, godziny oraz preferowanego stolika zaprezentowany został przez Rysunek 2.



Rysunek 2 Przykład rezerwacji POSBistro. Źródło [1]

2.1.2 Aplikacja mobilna POSBistro

Aplikacja mobilna dostępna jest na urządzeniach z systemami Android jak i iOS. Przyspiesza ona proces obsługi klienta zarówno w lokalu, jak i poza nim. Klienci mogą składać zamówienia bezpośrednio przy stolikach za pomocą tabletów lub smartfonów. Aplikacja umożliwia wprowadzanie zamówień, dodawanie modyfikacji oraz notatek dla kuchni. Interfejs aplikacji przedstawiający zamówienia przedstawia Rysunek 3.



Rysunek 3. Przykład aplikacji mobilnej POSBistro. Źródło [1]

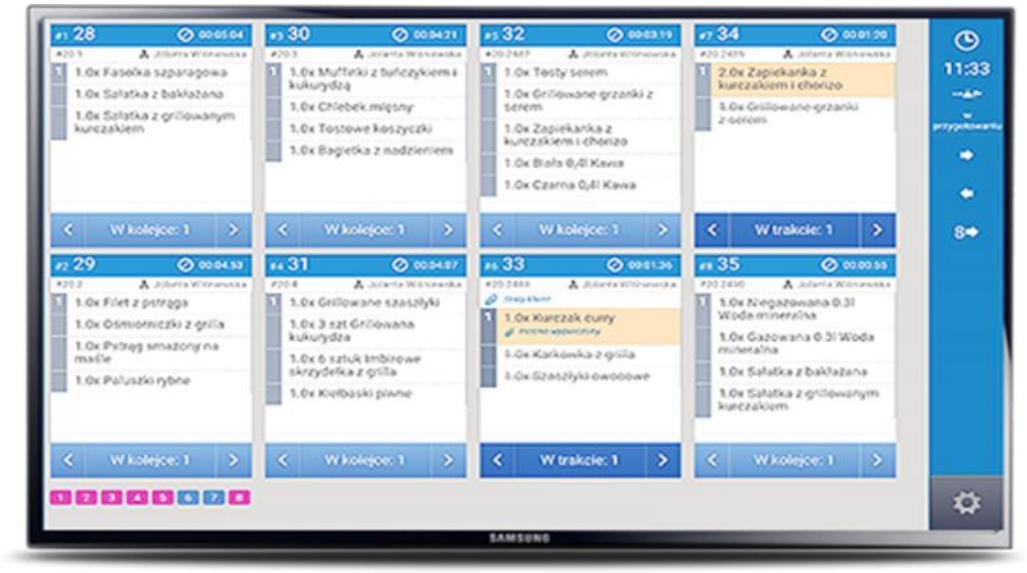
Aplikacja wspiera różne metody płatności w tym karty płatnicze, płatności mobilne oraz gotówkę. Proces ten zwiększa wygodę klientów oraz zmniejsza przypadki standardowych płatności, wewnątrz lokalu które dodatkowo obciążają zespół kelnerski.

System dodatkowo wspiera tworzenie i zarządzanie programami lojalnościowymi co pozwala na budowanie trwałych relacji z klientami i zachęca do ponownego skorzystania z usług danego lokalu jak i aplikacji.

2.1.3 Funkcjonalności POSBistro

System ten przede wszystkim skupia się na właścicielu restauracji, ułatwiając zarządzanie lokalem. Do podstawowych funkcji systemu POSBistro należą:

- Przyjmowanie zamówień klientów oraz wysłanie ich bezpośrednio do kucharza przy wykorzystaniu aplikacji na tablety i urządzenia mobilne. W ten sam sposób aplikacja umożliwia modyfikację istniejących już zamówień;
- W systemie dostępne jest realizowanie zamówień bezpośrednio przez Kucharza restauracji. Kucharz może zarządzać statusem zamówienia oraz natychmiast poinformować (poprzez urządzenie mobilne lub smartwatch) kelnera o wydaniu zamówienia. Panel kucharza przedstawia Rysunek 4;



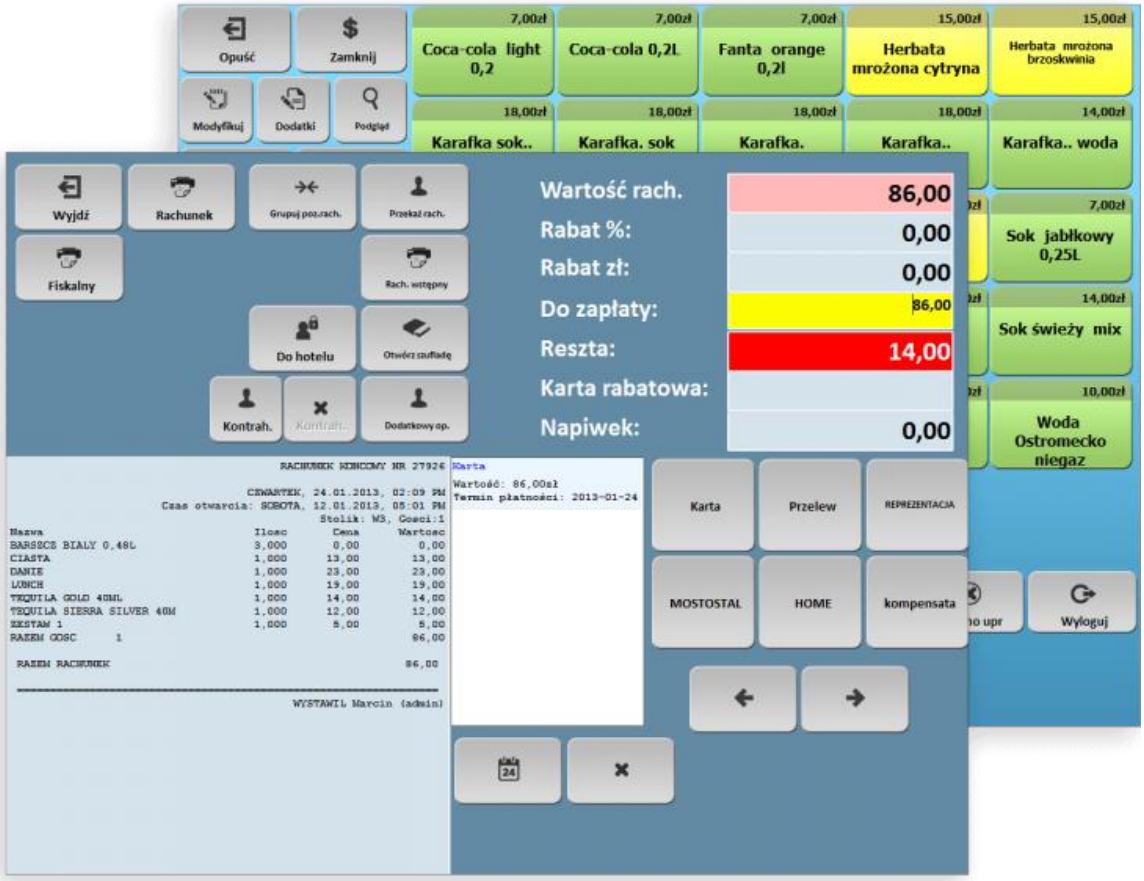
Rysunek 4 Przykład panelu kucharza. Źródło: [1]

- Monitorowanie Sprzedaży - Narzędzia analityczne, raporty sprzedażowe które pomagają właścicielom i menedżerom w monitorowaniu wyników finansowych, analizie najpopularniejszych pozycji menu, oraz ocenie efektywności pracowników;
- Zarządzanie Magazynem - Funkcje do zarządzania stanami magazynowymi, śledzenia zużycia produktów, generowania zamówień do dostawców oraz kontrolowania kosztów produktów;
- Bezpieczeństwo Danych - System zapewnia szyfrowanie danych co chroni przed utratą danych i nieautoryzowanym dostępem;
- Zarządzanie Menu - Umożliwia łatwe zarządzanie menu w tym dodawanie nowych pozycji, edytowanie istniejących, ustalanie cen oraz tworzenie promocji i rabatów.

2.2 X2System

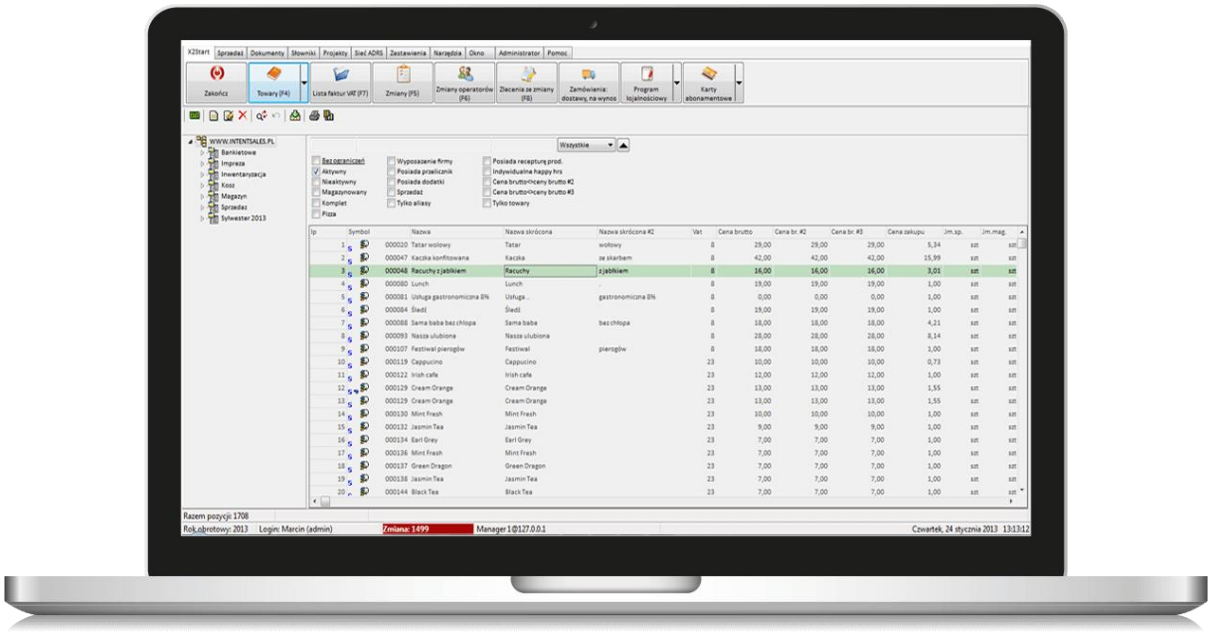
X2 [2] System to oprogramowanie stworzone z myślą o gastronomii i hotelarstwie. Obejmuje trzy główne moduły:

- X2 Kasa - dedykowany system obsługi punktów sprzedaży (POS), umożliwiający zarządzanie transakcjami, rachunkami oraz obsługą klientów. Posiada interfejs użytkownika i jest wyposażony w funkcje takie jak identyfikacja kelnera, obsługa różnych form płatności, zarządzanie menu, drukowanie zamówień czy raportowanie. X2 Kasa wspiera obsługę na ekranów dotykowych i może być uruchamiany w systemach MS Windows oraz Linux. POS systemu X2 Kasa został ukazuje Rysunek 5;



Rysunek 5 Przykład panelu X2Kasa. Źródło: [3]

- X2 Manager - służy do zarządzania restauracją, zapewniając pełną gospodarkę magazynową, rozliczenia finansowe i towarowe, a także raportowanie i analizy. Umożliwia tworzenie dokumentów, śledzenie zużycia surowców, kontrolę nad stanami magazynowymi oraz analizę finansową. Przykładowy panel systemu X2 Manager ukazuje Rysunek 6;



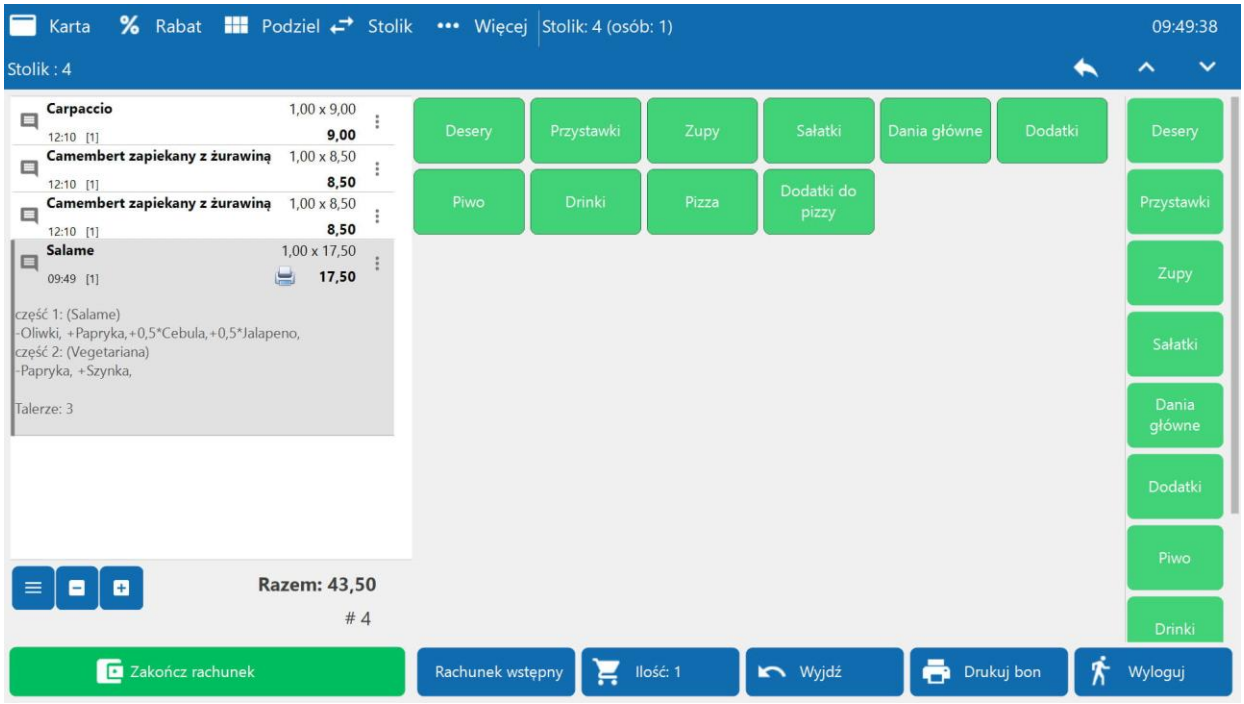
Rysunek 6 Przykład panelu zarządzania w aplikacji X2manager. Źródło: [3]

Dodatkowo, X2 System oferuje moduł X2 Sys pełniący rolę serwera wydruku fiskalnego i zarządzający rozchodami magazynowymi, oraz moduł X2 Raport umożliwiający generowanie różnorodnych raportów i analiz na podstawie zgromadzonych danych. System działa w środowisku Microsoft Windows, zarówno w sieci LAN, jak i na pojedynczych stanowiskach, co zapewnia elastyczność w zakresie konfiguracji i dostosowania do potrzeb użytkownika.

2.3 izzyRest

4Rest/izzyRest [5] to oprogramowanie stworzone specjalnie dla restauracji, zapewniające wszechstronną obsługę zarówno w obszarze punktów sprzedaży, jak i w kwestiach biurowych i magazynowych.

W obszarze punktów sprzedaży (POS) program oferuje zaawansowane funkcje, takie jak integrację z ekranami dotykowymi oraz obsługę terminali dotykowych, co ułatwia pracę kelnerom. Graficzna prezentacja sali restauracyjnej umożliwia intuicyjne zarządzanie stolikami, a obsługa drukarek kuchennych i rachunkowych sprawnie wspiera proces obsługi klienta. POS sytemu izzyRest przedstawia Rysunek 7.



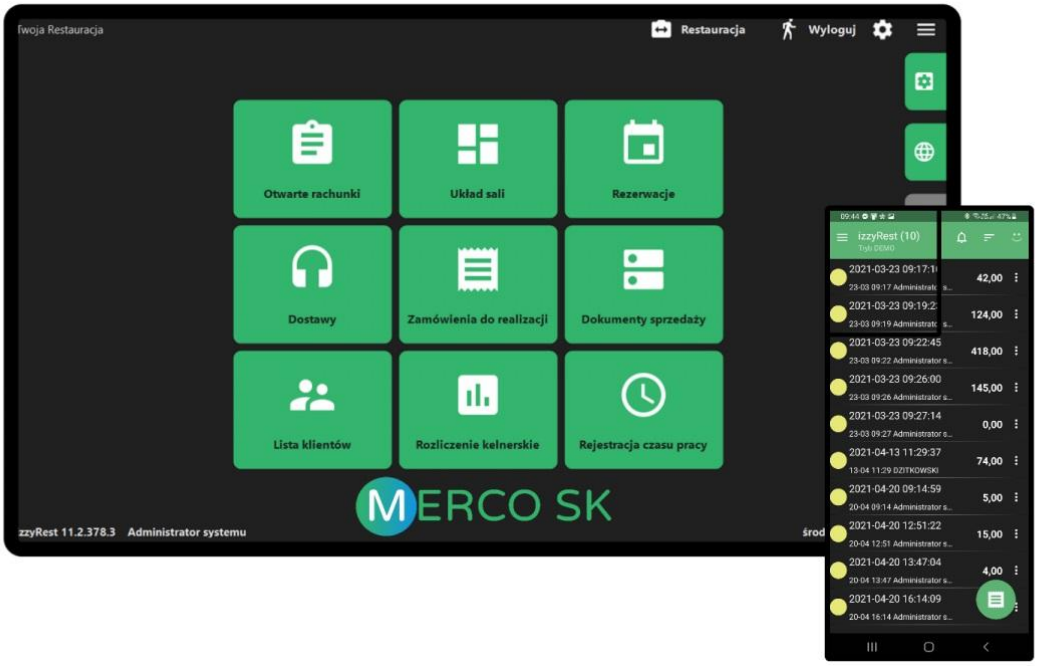
Rysunek 7 Przykład panelu pracownika izzyRest POS. Źródło:[5]

W kontekście funkcji biurowych, 4Rest/izzyRest zapewnia tworzenie i edycję towarów oraz receptur, a także definicję uprawnień użytkowników. Program generuje również różnorodne raporty i wykresy, ułatwiając analizę danych biznesowych restauracji.

Obsługa magazynów w trybie "*on-line*" umożliwia bieżące rozliczanie zapasów, a elastyczna struktura magazynów oraz operacji magazynowych dostosowuje się do potrzeb restauracji. Funkcje finansowe, takie jak obsługa paragonów i faktur VAT, a także generowanie danych do systemu księgowego, zapewniają kompleksowe wsparcie w zarządzaniu finansami.

Oprogramowanie umożliwia łatwą konfigurację interfejsu użytkownika, co pozwala na dostosowanie go do indywidualnych potrzeb restauracji. Dzięki możliwości pracy na zwykłych komputerach PC oraz zdalnej obsłudze przez Internet, 4Rest/izzyRest jest elastycznym i wszechstronnym narzędziem, które sprawnie wspiera restauracyjny biznes we wszystkich jego obszarach działania.

Dodatkowo, 4Rest/izzyRest oferuje wsparcie techniczne i aktualizacje, które gwarantują, że system pozostaje nowoczesny i zgodny z bieżącymi wymaganiami rynkowymi. Możliwość integracji z systemami płatności mobilnych oraz aplikacjami do rezerwacji online sprawia, że restauracje mogą dostarczać swoim klientom wygodne i nowoczesne rozwiązania. Ponadto, system oferuje moduły lojalnościowe, które pomagają w budowaniu trwałych relacji z klientami poprzez programy rabatowe i nagrody. Wszystko to czyni 4Rest/izzyRest kompleksowym rozwiązaniem, które podnosi efektywność operacyjną i zwiększa zadowolenie klientów. Główny panel programu izzyRest przedstawia Rysunek 8.

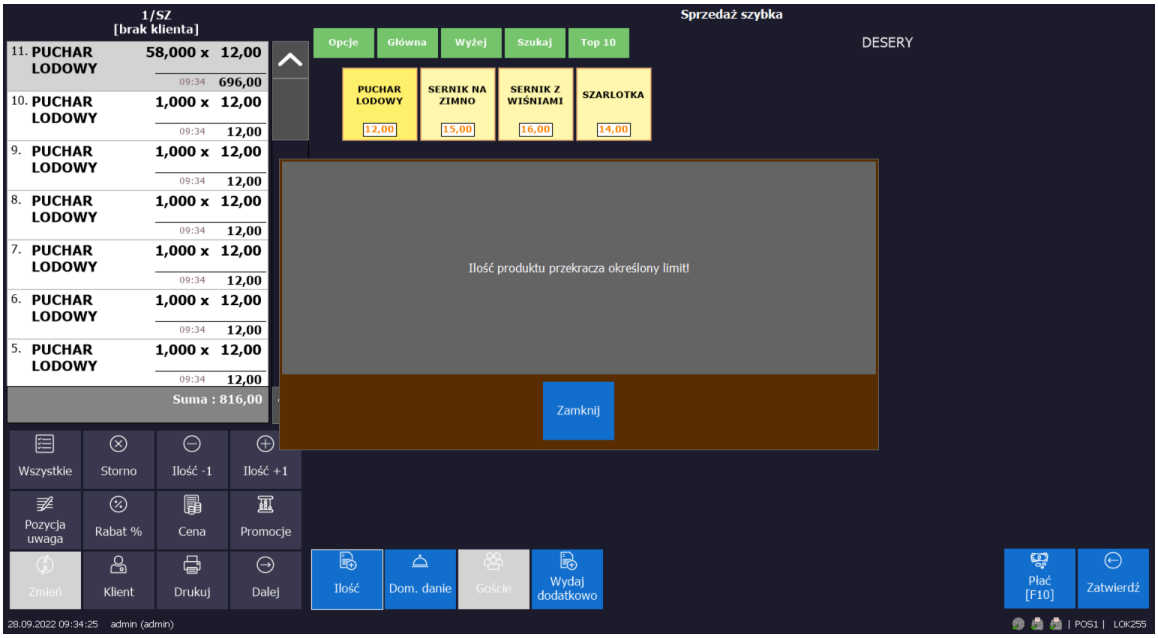


Rysunek 8 Przykład izzyRest. Źródło: [6]

2.4 FOODSOFT

FOODSOFT [7] to oprogramowanie dedykowane branży gastronomicznej, zaprojektowane w celu zarządzania lokalami takimi jak restauracje, kawiarnie i bary. Główne funkcje FOODSOFT obejmują zarządzanie zamówieniami, inwentaryzacją, personelem oraz magazynem.

Jednym z kluczowych elementów FOODSOFT jest moduł do zarządzania zamówieniami, który pozwala na łatwe i szybkie przyjmowanie zamówień zarówno na miejscu, jak i na wynos. System jest zintegrowany z nowoczesnymi terminalami POS, co umożliwia bezproblemową obsługę płatności i minimalizuje ryzyko błędów przy zamówieniach. POS systemu Foodsoft przedstawia Rysunek 9.

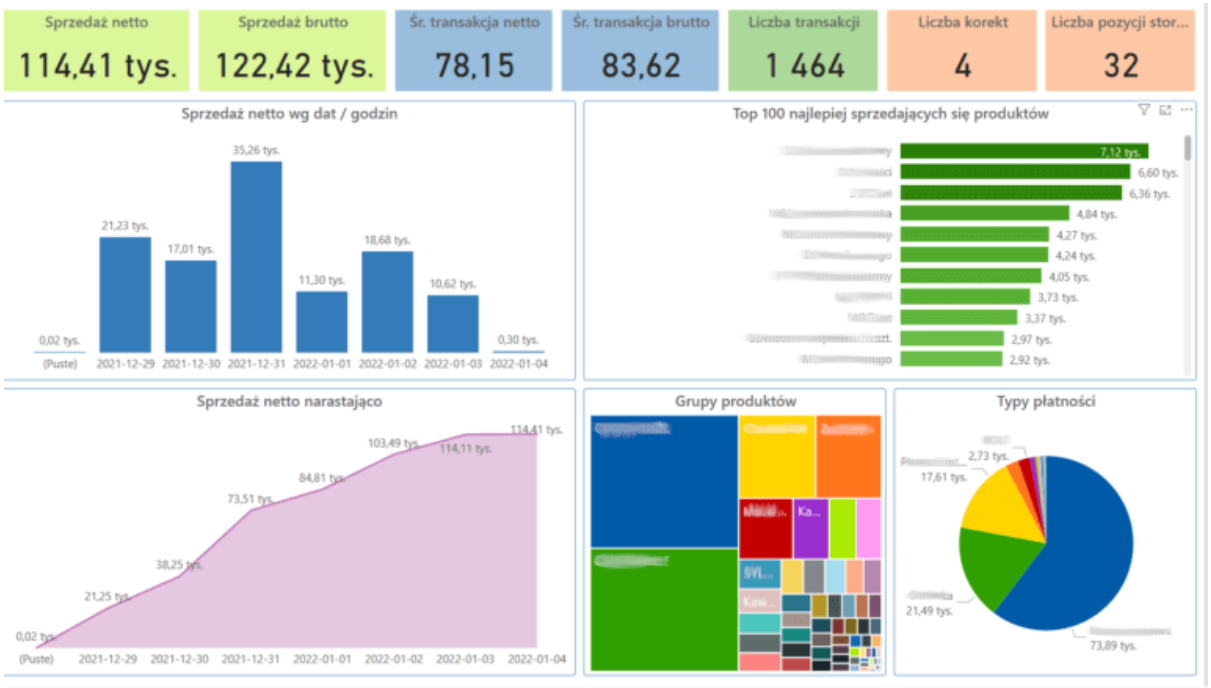


Rysunek 9 Przykład ekranu zarządzającego FOODSOFT. Źródło: [8]

Oprogramowanie zawiera również zaawansowane narzędzia do zarządzania inwentaryzacją. FOODSOFT monitoruje stany magazynowe, automatycznie aktualizując ilości produktów po każdym zamówieniu. Dzięki temu lokale mogą śledzić poziomy zapasów w czasie rzeczywistym co pozwala na optymalne zarządzanie zaopatrzeniem.

Kolejny istotny moduł to zarządzanie personelem. Oprogramowanie umożliwia tworzenie grafików, śledzenie godzin pracy pracowników oraz zarządzanie wynagrodzeniami. System może również automatycznie generować raporty wydajności, co pomaga w ocenie efektywności personelu.

System jest również wyposażony w funkcje analizy danych, co pozwala menedżerom na podejmowanie świadomych decyzji opartych na szczegółowych raportach i analizach trendów sprzedażowych (w podziale na sprzedaż bieżącą, tygodniową lub roczną) oraz preferencji klientów. Dzięki temu, FOODSOFT pomaga w optymalizacji procesów biznesowych oraz zwiększaniu rentowności lokalu. Aplikacja prezentuje dane w postaci tabelarycznej oraz przejrzystych wykresów tak jak na poniższym przykładzie. Statystyki dostępne w systemie Foodsoft prezentuje Rysunek 10.



Rysunek 10 Przykład panelu głównego FOODSOFT 5.3. Źródło: [9]

FOODSOFT integruje się z systemami rezerwacyjnymi, co umożliwia efektywne zarządzanie rezerwacjami stolików i zwiększa komfort klientów. Oprogramowanie oferuje również moduł do zarządzania relacjami z klientami (CRM), który pomaga w budowaniu lojalności i zwiększaniu powtarzalności wizyt. Dzięki możliwości wysyłania spersonalizowanych ofert oraz wiadomości marketingowych, FOODSOFT wspiera lokale w prowadzeniu skutecznych kampanii promocyjnych.

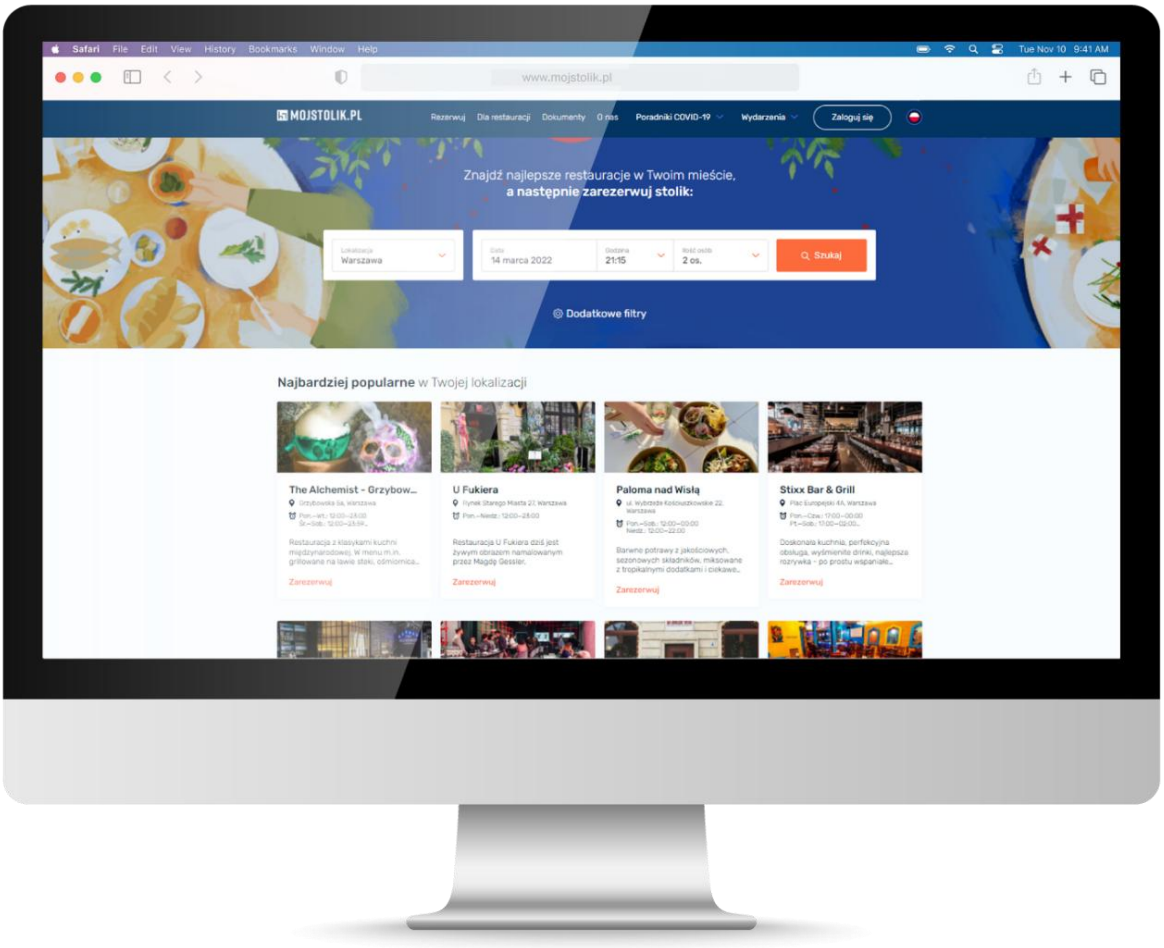
Całość oprogramowania jest łatwa w obsłudze i może być dostosowana do indywidualnych potrzeb każdego lokalu gastronomicznego, co czyni FOODSOFT wszechstronnym narzędziem wspierającym zarządzanie w branży gastronomicznej.

2.5 Mojstolik.pl

Według [10] aplikacja “MojStolik” to platforma rezerwacyjna dostępna dla warszawskich restauracji. Pozwala lokalom gastronomicznym na zarządzanie rezerwacjami stolików i integracje większości dostępnych kanałów komunikacji z gośćmi: telefon, email, aplikację mobilną MojStolik i

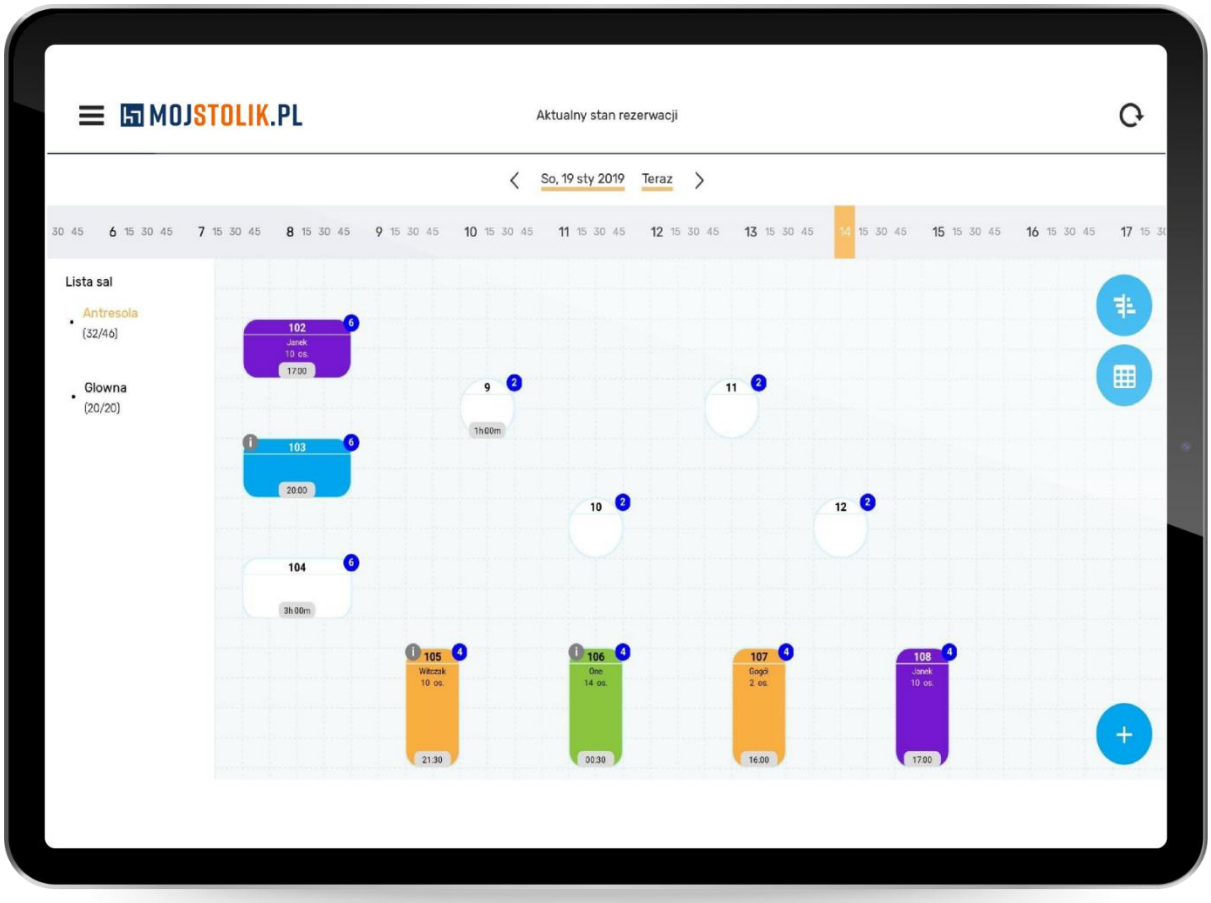
tzw. walk-in. Menedżerowie restauracji mają dostęp do dynamicznego widoku sali, sami decydują o układzie sal, liczbie dostępnych miejsc i na jak długo każdy stolik może być zarezerwowany.

System dostarcza również mechanizm potwierdzeń oraz przypomnień SMS. Rezerwacja odbywa się za pomocą systemu telefonicznego (IVR), który przyjmuje rezerwację zamiast zespołu lokalu, który w tym czasie może poświęcić się obsłudze gości w restauracji. Stronę główną strony internetowej Mojstolik.pl, która pozwala na rezerwacje stolika przedstawia Rysunek 11.



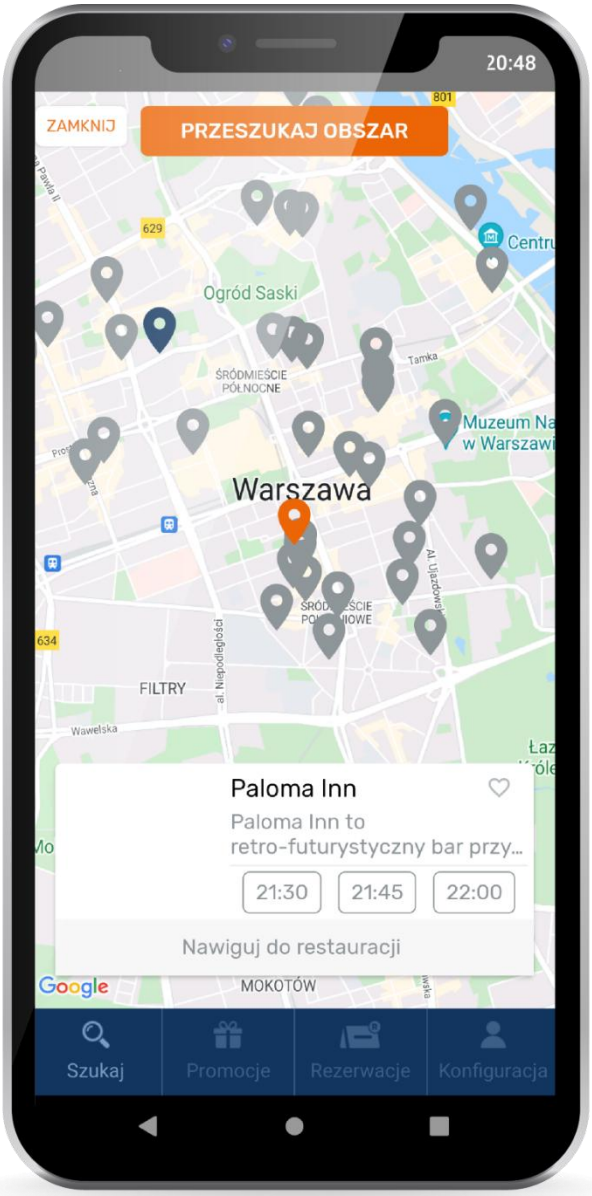
Rysunek 11 Przykład panelu zarządzania w aplikacji MójStolik. Źródło: [11]

Aplikacja jest wyposażona również w narzędzie wspomagające restauracje w budowaniu lojalności klientów. System umożliwia budowanie bazy danych gości, analizowanie historii rezerwacji oraz licznik wizyt pokazujący, który raz dana osoba odwiedza lokal. Goście podczas dokonywania rezerwacji mają możliwość dopisania uwag i preferencji. Przykładowy rozkład stolików na sali przedstawia Rysunek 12.



Rysunek 12Przykład rozkładu sali MójStolik. Źródło: [12]

MojStolik posiada również aplikację mobilną. Aplikacja umożliwia sprawdzenie dostępności i zarezerwowanie stolika na daną godzinę poprzez komunikację z restauracją w czasie rzeczywistym. Umożliwia dodanie do rezerwacji prośbę czy komentarz odnośnie do ulubionego stolika w lokalu, informacje o alergiach czy innych preferencjach. Restauracja za pomocą chatu może odpowiedzieć na daną prośbę. W aplikacji jest dostępne aktualne menu lokalu. Mapę z rozmieszczeniem restauracji w aplikacji mobilnej przedstawia Rysunek 13.

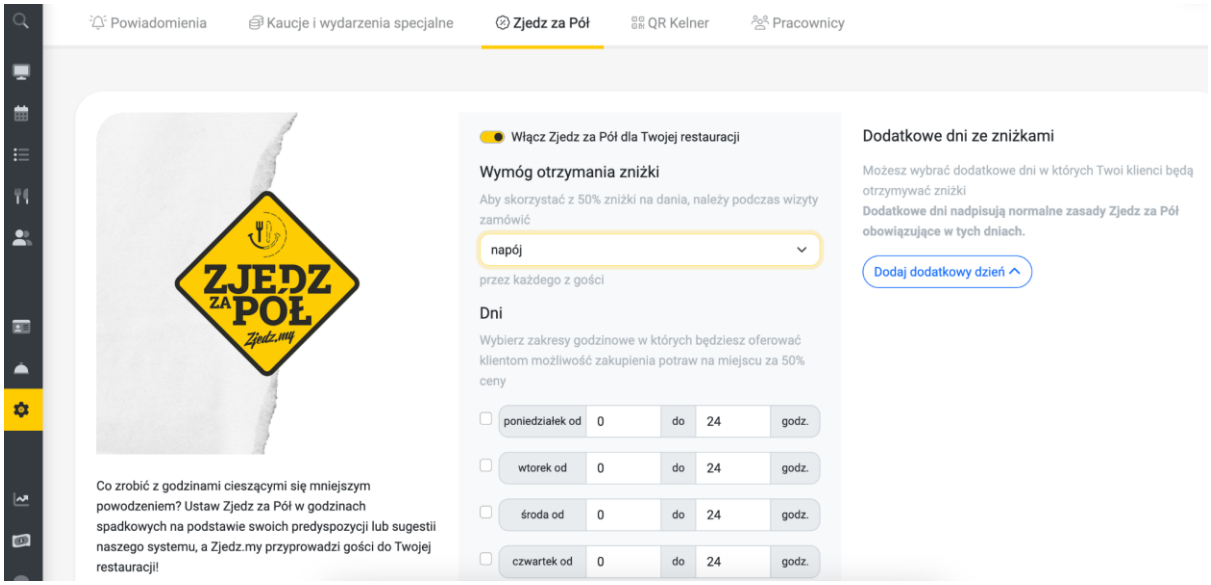


Rysunek 13 Przykład aplikacji mobilnej MójStolik. Źródło: [12]

2.6 Zjedz.my

Aplikacja Zjedz.my zgodnie z [13] to narzędzie do zarządzania rezerwacjami stolików w restauracjach, które oferuje szeroką gamę funkcji wspierających zarówno restauratorów, jak i ich gości. System umożliwia łatwe przyjmowanie rezerwacji, automatyzując wiele procesów, co pozwala zespołom restauracyjnym skoncentrować się na obsłudze gości. Restauracje mogą korzystać z integracji z systemami sprzedaży, co ułatwia monitorowanie rezerwacji oraz zamówień w jednym miejscu, a także efektywnie zarządzać obłożeniem lokalu, maksymalizując wykorzystanie dostępnych stolików.

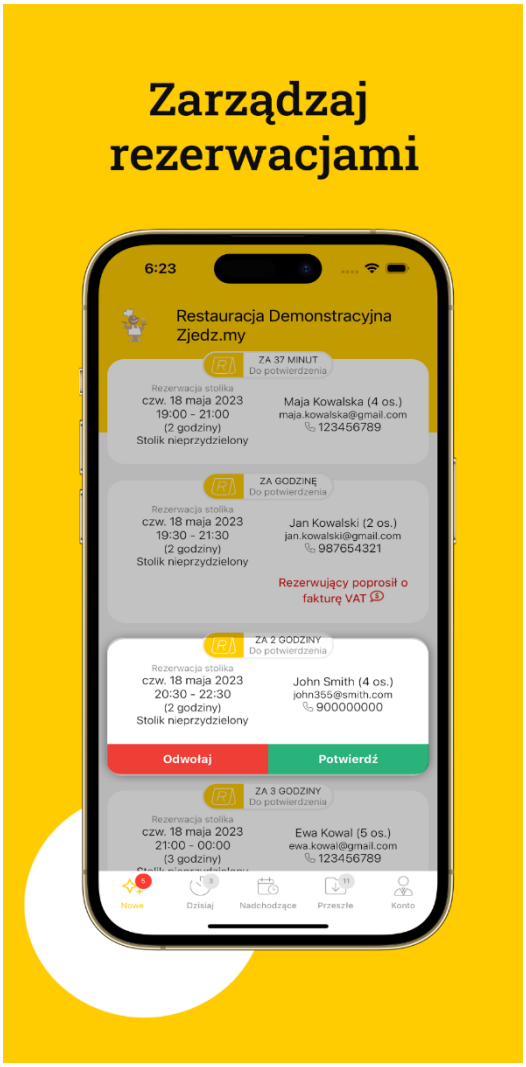
Zjedz.my wspiera również działania promocyjne restauracji dzięki narzędziom do marketingu, takim jak wysyłka wiadomości e-mail i SMS z przypomnieniami, ofertami specjalnymi czy promocjami. Aplikacja dostarcza rozbudowane raporty i analizy, które pomagają śledzić trendy, preferencje klientów oraz popularność poszczególnych pozycji w menu. Dzięki temu restauratorzy mogą podejmować bardziej świadome decyzje biznesowe. Rysunek 14 ukazuje możliwość włączenia programy lojalnościowego dla restauracji.



Rysunek 14 Przykład panelu ustawień Zjedz.my. Źródło: [14]

Goście z kolei mają możliwość szybkiego rezerwowania stolików online, z podglądem dostępności w czasie rzeczywistym. Aplikacja przypomina im o rezerwacjach, umożliwia ich modyfikację, a także oferuje dostęp do specjalnych promocji. Klienci mogą zostawiać opinie i recenzje, co pomaga innym użytkownikom w wyborze odpowiedniej restauracji.

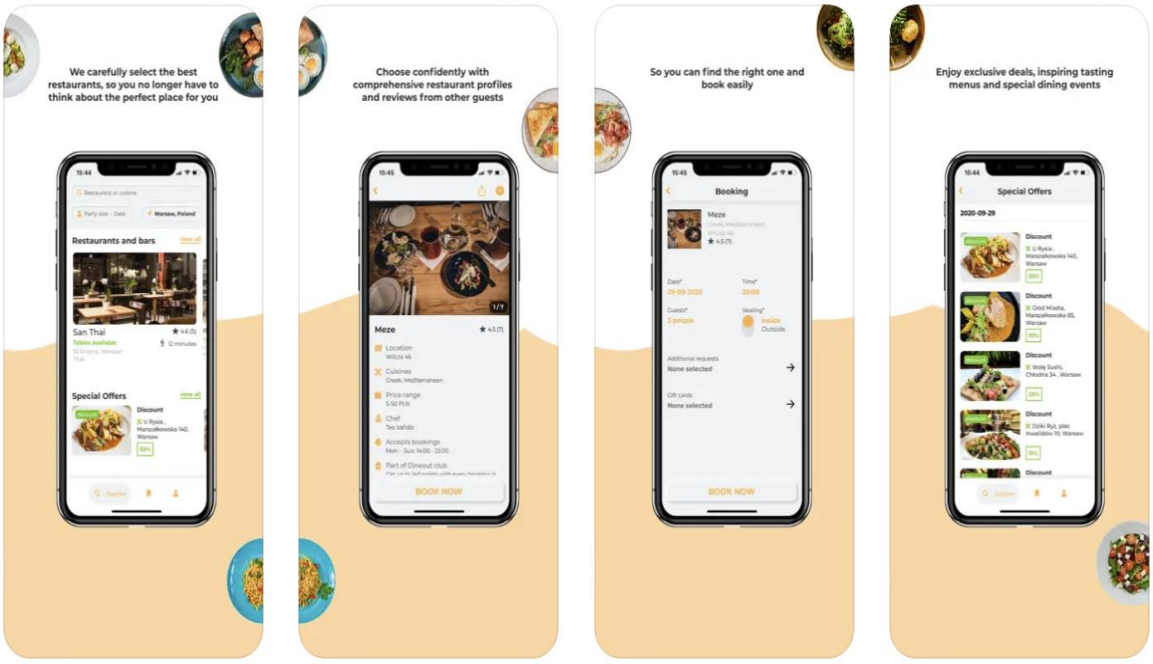
Zjedz.my oferuje również automatycznie generowane strony internetowe dla restauracji, które można dostosować do własnych potrzeb lub zintegrować z istniejącymi witrynami. Restauracje mają możliwość tworzenia menu w wersji cyfrowej z pomocą kodów QR, co pozwala na łatwe przeglądanie oferty przez gości na ich urządzeniach mobilnych. Menu może być automatycznie tłumaczone na różne języki, co zwiększa jego dostępność dla międzynarodowych klientów. System automatyzuje rezerwacje telefoniczne oraz pozwala na dokonywanie ich przez SMS, co dodatkowo upraszcza proces obsługi gości. Zarządzanie rezerwacjami z poziomu aplikacji ukazuje Rysunek 15.



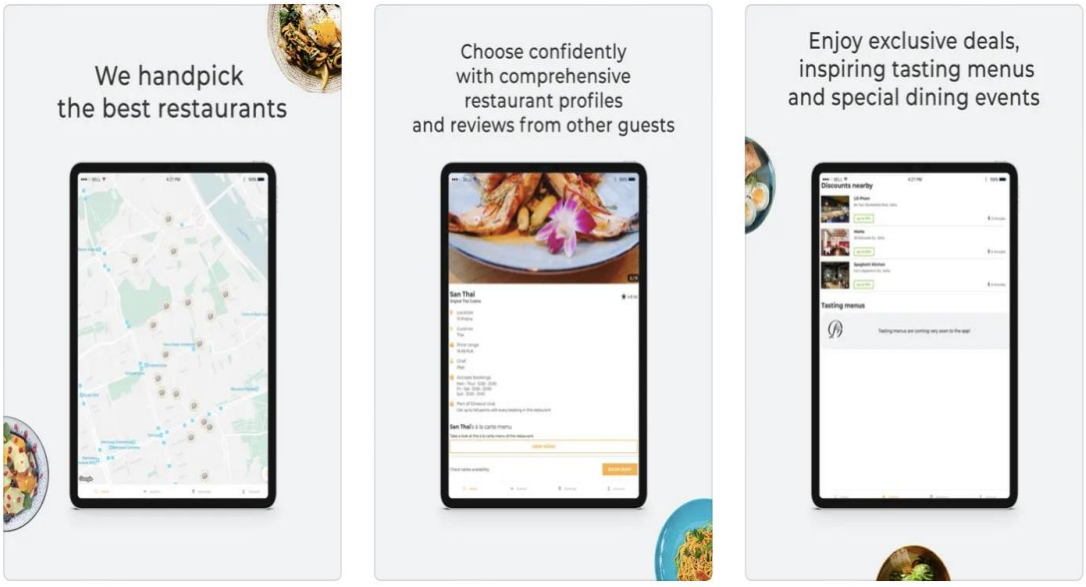
Rysunek 15 Przykład zarządzania restauracjami Zjedz.my. Źródło: [15]

2.7 Dineout.pl

Dineout.pl [16] to system pozwalający użytkownikom składać rezerwacje w wybranych restauracjach, za pomocą aplikacji webowej oraz aplikacja mobilnej na smartfona. Poniżej znajdują się zdjęcia poglądowe prezentujące wygląd aplikacji mobilnej.

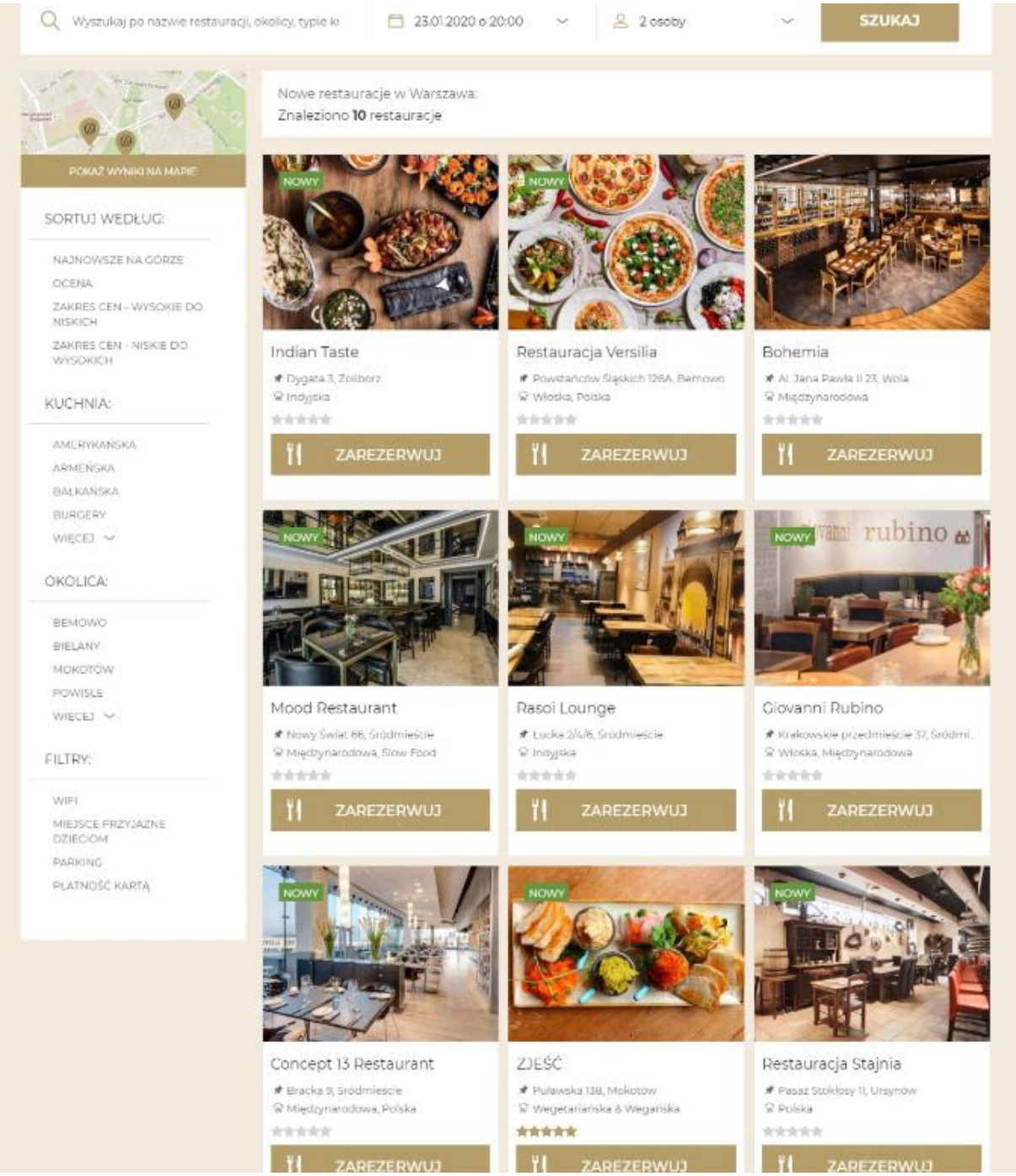


Rysunek 16 Aplikacja Dineout, panel użytkownika. Źródło: [17]



Rysunek 17 Aplikacja Dineout, panel pracownika. Źródło: [17]

Rysunek 18 jest przykładem, w którym użytkownicy mogą wyszukiwać lokale w panelu wyszukiwania lub na podstawie następujących kategorii: kuchni, okazji oraz polecanych lokali. W aplikacji znajduje się zakładka, w której można znaleźć oferty specjalne oraz zniżki na posiłki. Klienci mogą zakupić kartę podarunkową do wybranego lokalu oraz zapisać się do programu lojalnościowego "Dineout Club".

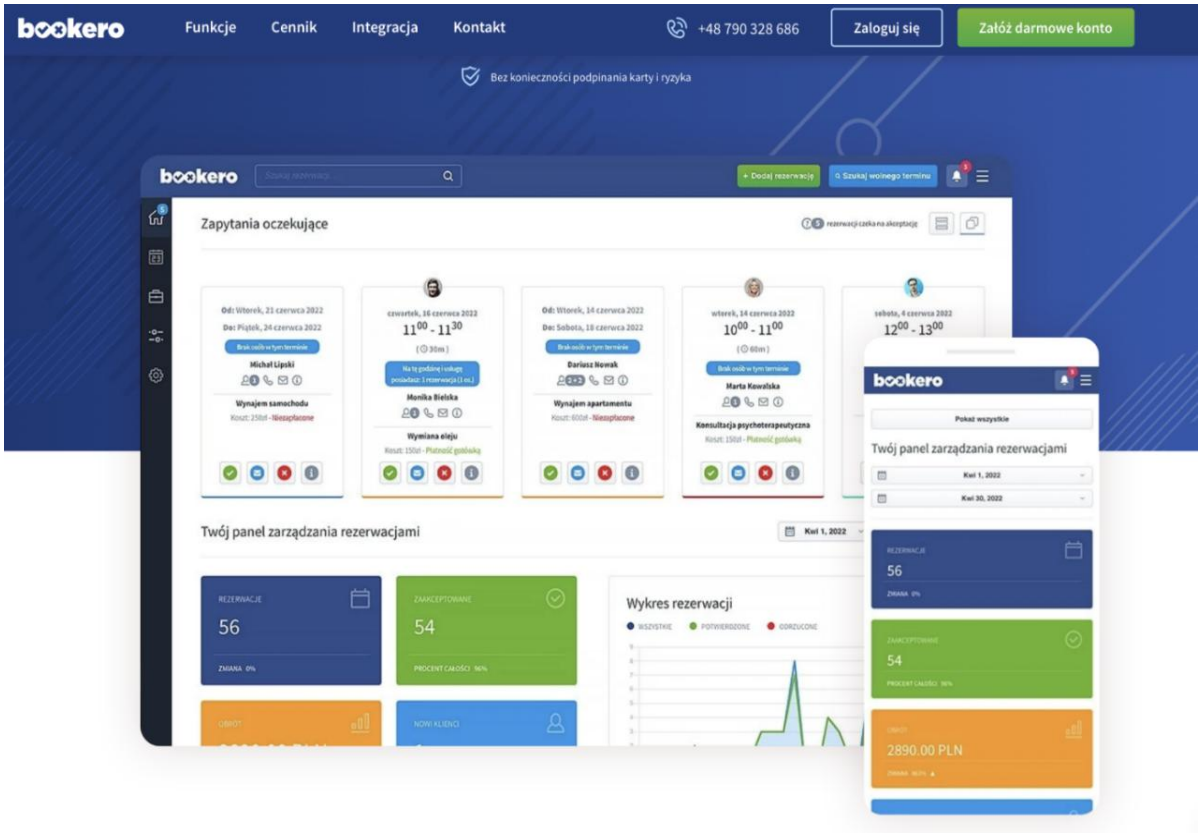


Rysunek 18 Przykład aplikacji web Dineout. Źródło: [18]

Dinout.pl oferuje również dodatkowe funkcje, takie jak możliwość zapisania ulubionych restauracji, przeglądanie zdjęć i menu, a także czytanie i dodawanie recenzji restauracji. Użytkownicy mogą również korzystać z promocji i specjalnych ofert dostępnych wyłącznie dla zarejestrowanych na platformie.

2.8 Bookero.pl

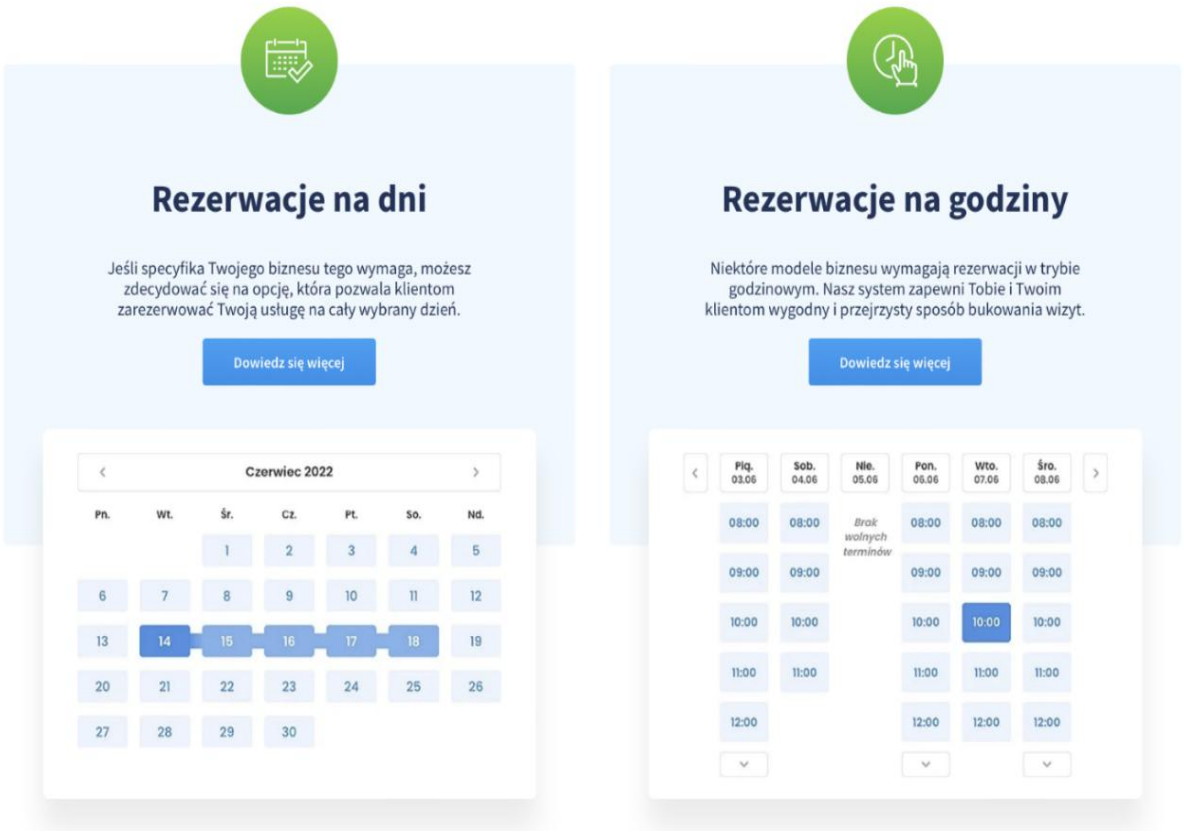
Bookero.pl [19] to aplikacja webowa stworzona do efektywnego i łatwego zarządzanie rezerwacjami online. Główny panel aplikacji Bookero przedstawia Rysunek 19.



Rysunek 19 Przykład aplikacji Bookero. Źródło: [19]

Z systemu mogą korzystać firmy specjalizujące się w różnych branżach m.in. wynajmu, medycznej, rozrywki, motoryzacji, a także gastronomicznej.

System oferuje szereg usług biznesowych dla firm klienckich, takich jak: określenie limitów i warunków rezerwacji, ustalenie cennika za usługi, cykliczne tworzenie usług, dodawanie rabatów na usługi czy płatność online. Użytkownicy aplikacji mogą zbudować swoją listę klientów, utworzyć grafik dostępności swoich pracowników oraz zintegrować stronę internetową swojej firmy z systemem. Wybór systemu rezerwacji ukazuj Rysunek 20.



Rysunek 20 Przykład panelu rezerwacji Bookero. Źródło: [19]

W celu zapewnienia bezpieczeństwa prawnego, system oferuję funkcjonalność, która zadba o zgodność oferowanych usług z RODO. Dodatkowo korzystając z aplikacji istnieje możliwość stworzenia spersonalizowanych powiadomień mailowych i rozsyłanie ich do klientów, a także możliwość tworzenia raportów.

Aplikacja pozwala na rezerwację usług na konkretne dni bądź godziny, w zależności od potrzeb biznesowych danej firmy. Dla swoich klientów Bookero.pl oferuje 14 dniowy okres próbny.

2.9 Podsumowanie

Opisane systemy zarządzania gastronomią wyróżniają się nowoczesnym podejściem do organizacji pracy w lokalach, łącząc różnorodne funkcje, które wspierają codzienne operacje, optymalizują procesy i ułatwiają obsługę klienta. Dzięki zaawansowanym rozwiązaniom technologicznym umożliwiają elastyczne zarządzanie zarówno rezerwacjami, jak i zamówieniami, jednocześnie zapewniając pełną kontrolę nad finansami i zapasami. Integracja z innymi narzędziami oraz możliwości dostosowania systemów do specyficznych potrzeb użytkowników stanowią istotny atut. Szczególnie interesujące są elementy związane z budowaniem relacji z klientami, jak programy lojalnościowe, które wspierają zaangażowanie i powrót gości.

Efektywność, skalowalność oraz przejrzystość działania to kluczowe aspekty, które pozwalają tym rozwiązaniom wspierać zarówno mniejsze lokale, jak i większe przedsiębiorstwa gastronomiczne. Dzięki szczegółowej analizie dostępnych na rynku programów oraz ich funkcjonalności, mogliśmy precyzyjnie określić, jakie cechy powinien posiadać nasz system, aby wyróżniać się na tle konkurencji. Zrozumienie mocnych stron istniejących rozwiązań pozwoliło nam nie tylko zidentyfikować kluczowe funkcjonalności, ale również dostrzec pewne braki, które mogą stać się przewagą naszego produktu.

3 Propozycja systemu Reservant

System Reservant ma na celu usprawnienie procesów związanych z funkcjonowaniem restauracji. Dzięki automatyzacji wielu zadań, system ten przyczyni się do poprawy jakości obsługi klientów, zwiększenia efektywności działania restauracji oraz odciążenia pracowników. W niniejszym rozdziale przedstawione zostaną między innymi wymagania funkcjonalne i niefunkcjonalne systemu Reservant, wsparte odpowiednimi diagramami przypadków użycia, diagramami stanów oraz diagramem klas.

3.1 Cele

Reservant jako cel obiera zautomatyzowanie i usprawnienie procesów zarządzania w restauracjach zarówno po stronie Właściciela lokalu, jak i Klientów. System ma na celu poprawę jakości usług, zwiększenie efektywności oraz ułatwienie zarządzania zasobami, co w konsekwencji przyczyni się do wzrostu satysfakcji klientów i rentowności restauracji. Dodatkowo aplikacja wspiera budowanie relacji międzyludzkich, ułatwiając wspólne planowanie spotkań oraz integrację w gronie znajomych, co czyni ją narzędziem nie tylko funkcjonalnym, ale również sprzyjającym rozwijaniu więzi społecznych.

3.2 Założenia

Reservant jest systemem, który wspomaga działalność różnych placówek gastronomicznych, dostosowując się do ich specyficznych potrzeb. W przypadku restauracji, system ten nie tylko ułatwia obsługę zamówień, ale również optymalizuje procesy zarządzania magazynem. Właściciele restauracji mają możliwość dodawania wydarzeń okolicznościowych które mają za zadanie przyciągnąć nowych potencjalnych klientów.

Po stronie klienta, aplikacja Reservant oferuje szereg funkcji, które mają na celu maksymalne ułatwienie korzystania z usług wybranej restauracji. Klienci mogą łatwo zarezerwować stolik za pomocą aplikacji, co eliminuje konieczność dzwonienia do restauracji. System potwierdza rezerwację natychmiastowo, co zapewnia wygodę i oszczędność czasu. Aplikacja prezentuje aktualne menu restauracji, umożliwiając klientom przeglądanie dań oraz napojów. Klienci mogą również składać zamówienia z wyprzedzeniem, co przyspiesza proces obsługi po przybyciu do lokalu.

Interfejs aplikacji został zaprojektowany z myślą o maksymalnej intuicyjności, co oznacza, że zarówno klienci, jak i pracownicy nie będą napotykać problemów podczas jego użytkowania. Dzięki przejrzystemu układowi i przyjaznej nawigacji, każdy użytkownik szybko odnajdzie najpotrzebniejsze w danym momencie funkcje.

3.3 Słownik pojęć

Podrozdział przedstawia zestaw regularnie używanych terminów w opisywanym systemie do określenia jego poszczególnych składowych.

- **Bar** - lokal gastronomiczny, który koncentruje się na serwowaniu napojów, głównie alkoholowych. Często oferuje również proste przekąski lub szybkie dania, takie jak burgery czy frytki. Zazwyczaj ma luźną atmosferę, a klienci mogą tu spędzać czas towarzysko;
- **Biuro obsługi klienta (BOK)** - dział lub punkt kontaktowy w firmie, który zajmuje się obsługą zapytań, problemów i potrzeb klientów. W lokalach gastronomicznych BOK może odpowiadać na pytania dotyczące rezerwacji, przyjmować uwagi czy skargi oraz informować klientów o menu i dostępnych usługach;
- **Klient** - osoba odwiedzająca lokal gastronomiczny w celu skorzystania z jego oferty, tj. zjedzenia posiłku, zamówienia napoju lub wzięcia udziału w wydarzeniu. Klient może

wyrażać swoje opinie, zgłaszać uwagi oraz wystawiać recenzje, które wpływają na reputację lokalu;

- **Lokal gastronomiczny** - ogólne określenie miejsc oferujących jedzenie i napoje, takich jak restauracje, bary, kawiarnie czy food trucki. Lokale gastronomiczne różnią się charakterem, specjalizacją kulinarną, cenami i atmosferą, dostosowując się do różnych preferencji klientów;
- **Lokalizacja** - fizyczne położenie lokalu gastronomicznego. Lokalizacja ma wpływ na liczbę odwiedzających, dostępność dla klientów oraz prestiż. W przypadku restauracji w centrach miast lub w atrakcyjnych turystycznie miejscach jest szczególnie ważna dla pozyskania klientów;
- **Menu** - spis oferowanych w lokalu potraw, napojów i deserów wraz z cenami. Menu jest ważnym elementem każdego lokalu gastronomicznego, odzwierciedlającym jego charakter, specjalizację kuchenną oraz zakres cenowy. Dobrze przygotowane menu często zawiera również informacje o składnikach lub opcjach dietetycznych;
- **Pracownik zaplecza** - osoba odpowiedzialna za zadania wykonywane poza bezpośrednią obsługą klienta, tj. przygotowywanie posiłków w kuchni, sprzątanie czy prace magazynowe. Pracownicy zaplecza dbają o jakość i terminowość realizacji zamówień oraz wspierają dział obsługi klienta, zapewniając sprawne działanie lokalu;
- **Recenzja** - opinia wyrażona przez klienta po wizycie w lokalu, opublikowana np. w internecie, na portalach społecznościowych lub stronach z ocenami. Recenzje mogą być pozytywne lub negatywne i często wpływają na reputację lokalu oraz zainteresowanie nim innych klientów;
- **Rezerwacja** - zarezerwowanie miejsca lub stolika w lokalu gastronomicznym na określony czas. Rezerwacje są często przyjmowane przez restauracje, aby zapewnić gościom miejsce i uniknąć kolejek. Można je zrealizować telefonicznie, online lub osobiście;
- **Restauracja** - lokal gastronomiczny, który oferuje pełne posiłki, od przystawek po desery, oraz profesjonalną obsługę kelnerską. Restauracje często mają rozbudowane menu i większy wybór dań oraz napojów, co odróżnia je od barów czy kawiarni;
- **Zamówienie** - prośba klienta o przygotowanie konkretnego dania lub napoju, składana bezpośrednio u obsługi lub poprzez aplikację. Zamówienie obejmuje wybór pozycji z menu i określenie szczegółowych preferencji, np. sposobu przygotowania lub dodania wybranych składników;
- **Zgłoszenie** - forma kontaktu klienta z obsługą, mająca na celu zgłoszenie problemu, zapytania lub uwagi dotyczącej świadczonej usługi. Zgłoszenia mogą dotyczyć kwestii takich jak błędy w zamówieniu, zastrzeżenia co do jakości czy uwagi związane z rezerwacją.

3.4 Użytkownicy systemu

Fragment przedstawia poszczególnych użytkowników omawianego systemu w szczegółowy oraz techniczny sposób pozwalający zidentyfikować rolę jaką odgrywają.

- Klient;
    - Przechowywane informacje;
        - Imię,
        - Nazwisko,

- Login,
- Hasło,
- Zdjęcie profilowe,
- Data rejestracji,
- Data urodzenia,
- Reputacja,
- Email,
- Numer telefonu,
- Wiek,
- Stan portfela.

- Dostępne funkcjonalności;
  - Możliwość wyszukania barów w obszarze uzależnionym od ustalonego czasu dotarcia oraz ilości potrzebnych miejsc,
  - Możliwość rezerwacji baru na określony czas wraz z wpłatą kaucji, ewentualnym złożeniem zamówienia,
  - Możliwość wystawienia opinii i oceny lokalu,
  - Możliwość edycji oceny i opinii dla lokalu,
  - Możliwość udostępnienia lokalizacji znajomym,
  - Możliwość zgłoszenia incydentu do administratora (np. Naruszenie regulaminu aplikacji),
  - Możliwość zgłoszenia chęci usunięcia konta wraz z zachowaniem prawa do bycia zapomnianym (RODO art. 17 ust. 1),
  - Utworzenie wydarzenia,
  - Możliwość przeglądania wydarzeń stworzonych przez innych użytkowników, z możliwością filtrowania,
  - Możliwość zapisania rezerwacji z historii rezerwacji na liście “szybkie rezerwacje”,
  - Możliwość utworzenia nowej szybkiej rezerwacji i dodanie jej na liście “szybkie rezerwacje”,
  - Możliwość uzupełnienia profilu dodatkowymi informacjami tj. opis, zdjęcie, inne,
  - Możliwość dodania znajomego,
  - Możliwość wysłania wiadomości do innego klienta,
  - Możliwość dodania wydarzenia na tablicy wydarzeń,
  - Możliwość doładowania portfela wykorzystywanego przy składaniu rezerwacji lub zamawianiu jedzenia i picia,
  - Możliwość wyświetlenia historii płatności,

        - Możliwość wyświetlenia historii wydarzeń, w których brał udział,
        - Możliwość zamieszczenia wydarzenia z historii w zakładce dzieje się oraz dodania do niego zdjęć i opisu,
        - Możliwość utworzenia wątku z innym użytkownikiem i wątków grupowych,
        - Możliwość komentowania postów,
        - Możliwość przełączenia się między trybem ciemnym i jasnym,
        - Możliwość zmiany języka aplikacji (na początek polski i angielski),
        - przeglądanie opinii wybranego lokalu,
- Właściciel lokalu;
    - Przechowywane informacje;
        - Imię,
        - Nazwisko,
        - Login,
        - Hasło,
        - Zdjęcie profilowe,
        - Data rejestracji,
        - Email,
        - Data urodzenia,
        - Dowód osobisty,
        - Flaga dowodu,
        - Numer telefonu.
    - Dostępne funkcjonalności;
        - Możliwość wyświetlenia i odpowiedzi na opinie użytkowników,
        - Możliwość wyświetlenia wszystkich zamówień z wybranego przedziału czasu,
        - Możliwość wyświetlania statystyk z zadanego okresu,
        - Możliwość edycji oferty oraz odpowiadającego jej cennika z opcją dodawania zdjęć jedzenia i napojów,
        - Możliwość dodawania menu sezonowego i promocji dla lokalu,
        - Możliwość modyfikowania ceny kaucji za rezerwację oraz możliwych czasów rezerwacji,
        - Możliwość rejestracji lokalu w aplikacji,
        - Możliwość dodania lokalu do grupy lokali przy lub po rejestracji,
        - Możliwość wyboru funkcjonalności dla lokalu tj. system rezerwacji, system zamawiania przez aplikacje do stolika, dostawy itp.,
        - wygenerowanie listy zakupów zawierającej rzeczy niezbędne do uzupełnienia zapasów,

        - Możliwość wglądu w stan magazynowy zaplecza,
        - Możliwość wygenerowania listy zakupów,
        - Możliwość edycji listy zakupów,
        - Możliwość zatwierdzenia listy zakupów,
        - Możliwość wglądu w zgłoszenia dotyczące jego pracowników,
        - Możliwość utworzenia konta dla pracownika,
        - Możliwość edytowania uprawnień pracowników, w zakresie dostępu do;
            - możliwości podglądu zajętych stolików oraz modyfikowania ilości miejsc dostępnych dla użytkowników aplikacji,
            - możliwości tymczasowego wycofania produktu z oferty,
            - możliwości korekty stanu magazynowego zaplecza,
            - możliwości edycji kopii listy zakupów, w celu wprowadzenia stanu faktycznego dostawy do systemu,
            - możliwości zatwierdzenia zmodyfikowanej listy inwentaryzacyjnej,
            - możliwość nadawania pracownikom uprawnień do publikowania komentarzy w zakładce “dzieje się”,
        - Możliwość przełączenia się między trybem ciemnym i jasnym,
        - Możliwość zmiany języka aplikacji (na początek polski i angielski),
        - przeglądanie opinii wybranego lokalu,
- Pracownik Sali;
    - Przechowywane informacje;
        - Imię,
        - Nazwisko,
        - Login,
        - Hasło,
        - Zdjęcie profilowe,
        - Data rejestracji,
        - Email,
        - Data urodzenia,
        - Dowód osobisty,
        - Flaga dowodu,
        - Numer telefonu.
    - Dostępne funkcjonalności:
        - Możliwość ręcznej zmiany statusu stolika,

        - Możliwość podglądu zajętych stolików oraz modyfikowania ilości miejsc dostępnych dla użytkowników aplikacji,
        - Możliwość anulowania rezerwacji,
        - Możliwość wyświetlenia opinii klientów i gości,
        - Możliwość wyświetlenia zamówień, które obecnie obsługuje,
        - Możliwość obsługi zgłoszonych rezerwacji (akceptacja, odrzucenie, zmiana numeru stolika przypisanego do rezerwacji),
        - Możliwość zaznaczenia w ramach rezerwacji, że klient zabrał resztę zamówienia na wynos,
        - Możliwość zgłoszenia skargi na klienta,
        - Możliwość przełączenia się między trybem ciemnym i jasnym,
        - Możliwość zmiany języka aplikacji (na początek polski i angielski),
        - przeglądanie opinii wybranego lokalu,
- Pracownik zaplecza;
    - Przechowywane informacje;
        - Imię,
        - Nazwisko,
        - Login,
        - Hasło,
        - Zdjęcie profilowe,
        - Data rejestracji,
        - Email,
        - Data urodzenia,
        - Dowód osobisty,
        - Flaga dowodu,
        - Numer telefonu.
    - Dostępne funkcjonalności:
        - Możliwość wyświetlenia opinii klientów i gości,
        - Możliwość tymczasowego wycofania produktu z oferty (np. skończyły się składniki),
        - Możliwość wyświetlenia zamówień, które obecnie obsługuje,
        - Możliwość zmiany statusu zamówienia (na wszelki wypadek np. błąd systemu),
        - Możliwość zmiany statusu konkretnej pozycji w zamówieniu,
        - Możliwość korekty stanu magazynowego zaplecza,
        - Możliwość wyświetlenia stanu magazynowego zaplecza,

        - Możliwość zgłoszenia zapotrzebowania na jakiś składnik. Składnik zostanie dodany do listy zmian do akceptacji przez właściciela lokalu,
        - Możliwość edycji kopii listy zakupów, w celu wprowadzenia stanu faktycznego dostawy do systemu. Kopia przyjmuje rolę listy inwentaryzacyjnej do odbioru dostawy,
        - Możliwość zatwierdzenia listy inwentaryzacyjnej,
        - Możliwość zgłoszenia skargi na klienta,
        - Możliwość przełączenia się między trybem ciemnym i jasnym,
        - Możliwość zmiany języka aplikacji (na początek polski i angielski),
        - przeglądanie opinii wybranego lokalu,
- Kierownik BOK;
    - Przechowywane informacje;
        - Imię,
        - Nazwisko,
        - Login,
        - Hasło,
        - Zdjęcie profilowe,
        - Data zatrudnienia,
        - Pensja.
    - Dostępne funkcjonalności:
        - Możliwość wyświetlania zgłoszeń właścicieli i użytkowników,
        - Możliwość przyznania kodu promocyjnego,
        - Możliwość rozpatrzenia zgłoszenia,
        - Możliwość przekazania zgłoszenia kierownikowi BOK,
        - Możliwość przekazania zgłoszenia technicznego administratorowi,
        - Możliwość wyświetlania informacji o zalogowanym użytkowniku, którego dotyczy zgłoszenie,
        - Możliwość wyświetlania informacji o zalogowanym użytkowniku, który utworzył zgłoszenie,
        - Możliwość wyświetlania informacji o pracowniku, który utworzył zgłoszenie,
        - Możliwość wyświetlania informacji o zamówieniu, którego dotyczy zgłoszenie,
        - Możliwość wyświetlenia historii płatności klienta powiązanego z rozpatrywanym zgłoszeniem,
        - Możliwość wyświetlenia informacji o lokalu, które są dostępne w systemie, powiązanego z rozpatrywanym zgłoszeniem,
        - Możliwość zablokowanie/odblokowania użytkownika lub właściciela,

        - Możliwość potwierdzenia przyjęcia lokalu,
        - Możliwość przełączenia się między trybem ciemnym i jasnym,
        - Możliwość zmiany języka aplikacji (na początek polski i angielski),
        - przeglądanie opinii wybranego lokalu,
- Pracownik BOK;
    - Przechowywane informacje;
        - Imię,
        - Nazwisko,
        - Login,
        - Hasło,
        - Zdjęcie profilowe,
        - Data zatrudnienia,
        - Pensja.
    - Dostępne funkcjonalności:
        - Możliwość wyświetlania zgłoszeń właścicieli i użytkowników,
        - Możliwość przyznania kodu promocyjnego,
        - Możliwość rozpatrzenia zgłoszenia,
        - Możliwość przekazania zgłoszenia kierownikowi BOK,
        - Możliwość przekazania zgłoszenia technicznego administratorowi,
        - Możliwość wyświetlania informacji o zalogowanym użytkowniku, którego dotyczy zgłoszenie,
        - Możliwość wyświetlania informacji o zalogowanym użytkowniku, który utworzył zgłoszenie,
        - Możliwość wyświetlania informacji o pracowniku, który utworzył zgłoszenie,
        - Możliwość wyświetlania informacji o zamówieniu, którego dotyczy zgłoszenie,
        - Możliwość wyświetlenia historii płatności klienta powiązanego z rozpatrywanym zgłoszeniem,
        - Możliwość wyświetlenia informacji o lokalu, które są dostępne w systemie, powiązanego z rozpatrywanym zgłoszeniem,
        - Możliwość przełączenia się między trybem ciemnym i jasnym,
        - Możliwość zmiany języka aplikacji (na początek polski i angielski),
        - przeglądanie opinii wybranego lokalu,
- Gość;
    - Dostępne funkcjonalności;
        - Możliwość zarejestrowania konta,

- Możliwość zalogowania się na już istniejące konto,
- Możliwość wyświetlenia mapy barów z możliwością konfiguracji wyświetlania,
- Możliwość przeglądania opinii wybranego lokalu.

3.5 Wymagania funkcjonalne

Tabela 11 Spis wymagań funkcjonalnych. Źródło: Opracowanie własne

Lp.	Nazwa	Opis	Aktorzy	Warunki początkowe
1.	Rejestracja/ logowanie/ wylogowanie z systemu	Gość ma możliwość rejestracji w systemie	Gość	
2.	Logowanie/ wylogowanie z systemu	Gość ma możliwość zalogowania się do systemu/ wylogowania się z systemu	Gość	Użytkownik posiada konto
3.	Przeglądanie opinii wybranego lokalu	Gość ma możliwość przeglądania opinii o lokalu	Gość	
4.	Wyświetlenie mapy barów z możliwością konfiguracji wyświetlania	Wyświetlanie mapy, na której widoczne są lokalizacje lokali restauracji/ barów	Gość, Klient	
5.	Złożenie rezerwacji	Zalogowany	Klient, Pracownik BOK, Kierownik BOK	Użytkownik jest zalogowany do systemu
6.	Złożenie szybkiej rezerwacji		Klient, Pracownik BOK, Kierownik BOK	Użytkownik jest zalogowany do systemu
7.	Wystawienie opinii i oceny lokalu	Po wizycie w lokalu użytkownik może wystawić ocenę restauracji/ barowi	Klient, Pracownik BOK, Kierownik BOK	Po zakończonej wizycie w lokalu

8.	Wyświetlanie historii zamówień	Użytkownik ma możliwość przeglądania historii swoich zamówień	Klient, Pracownik BOK, Kierownik BOK	Użytkownik jest zalogowany do systemu
9.	Wyrażenie chęci usunięcia konta wraz z zachowaniem prawa do bycia zapomnianym	Użytkownik ma prawo zgłoszenie prośby o usunięcie konta wraz z zachowaniem prawa do bycia zapomnianym	Klient, Pracownik BOK, Kierownik BOK	Użytkownik posiada konto w systemie jako “Klient”
10.	Zgłoszenie skargi na pracownika	Użytkownik ma możliwość zgłoszenia skargi na pracownika lokalu	Klient, Pracownik BOK, Kierownik BOK	Po zakończonej wizycie w lokalu
11.	Zgłoszenie sprawy związanej z wizytą w lokalu	Użytkownik ma prawo wysłania zgłoszenia związanego z wizytą w lokalu (np. dotyczącego zgubionego przedmiotu)	Klient, Pracownik BOK, Kierownik BOK	Użytkownik posiada rezerwację w lokalu
12.	Utworzenie wydarzenia	Użytkownik może utworzyć wydarzenie, które dotyczy rezerwacji w danym lokalu. W wydarzeniu mogą brać inni użytkownicy	Klient, Pracownik BOK, Kierownik BOK	Użytkownik posiada rezerwację w lokalu
13.	Utworzenie wątku	Użytkownik może utworzyć konwersację, która posiada temat i może posiadać wielu uczestników	Klient, Pracownik BOK, Kierownik BOK	Użytkownik jest zalogowany w systemie
14.	Wyświetlanie wydarzeń stworzonych przez innych użytkowników z możliwością filtrowania	Użytkownik może spersonalizowanie wyświetlać wydarzenia na wizytę w lokalu, utworzone przez innych użytkowników	Klient, Pracownik BOK, Kierownik BOK	Użytkownik zalogowany jako “Klient”
15.	Rejestracja lokalu	Użytkownik może zarejestrować nową restaurację w systemie	Właściciel lokalu, Pracownik BOK, Kierownik BOK	Restauracja nie jest jeszcze zarejestrowana w systemie

16.	Możliwość edycji oferty oraz odpowiadającego jej cennika z opcją dodawania zdjęć jedzenia i napojów	Użytkownik może dodawać i edytować oferty menu dla restauracji z opcją dodawania zdjęć	Właściciel lokalu, Pracownik BOK, Kierownik BOK	Użytkownik posiada zarejestrowany w systemie lokal
17.	Dodawanie/ edytowanie uprawnień pracowników	Użytkownik może dodawać i zmieniać uprawnienia pracownikom	Właściciel lokalu, Pracownik BOK, Kierownik BOK	Użytkownik posiada zarejestrowanych pracowników w lokalu
18.	Wyświetlanie historii zamówień z zadanego okresu	Użytkownik może sprawdzić historię zamówień dla lokalu z wybranego okresu	Właściciel lokalu, Pracownik BOK, Kierownik BOK	Użytkownik posiada zarejestrowany w systemie lokal
19.	Wyświetlanie ocen i komentarzy dla lokalu	Użytkownik może wyświetlić oceny i komentarze “Klientów” dodane dla lokalu	Właściciel lokalu, Pracownik BOK, Kierownik BOK	Użytkownik posiada zarejestrowany w systemie lokal
20.	Wyświetlanie statystyk z zadanego okresu	Użytkownik może wyświetlić statystyki ze sprzedaży dla lokalu z danego okresu	Właściciel lokalu, Pracownik BOK, Kierownik BOK	Użytkownik posiada zarejestrowany w systemie lokal
21.	Wygenerowanie listy zakupów w oparciu o braki	Użytkownik może wygenerować listę produktów do zakupu w oparciu o braki w stanie magazynu dla konkretnego lokalu	Właściciel lokalu, Pracownik BOK, Kierownik BOK	Użytkownik posiada zarejestrowany w systemie lokal
22.	Zarządzanie listą zakupów	Użytkownik może zarządzać listą zakupów wygenerowaną na podstawie braków w magazynie dla konkretnego lokalu	Właściciel lokalu, Pracownik BOK, Kierownik BOK	Wygenerowanie listy zakupów w oparciu o braki

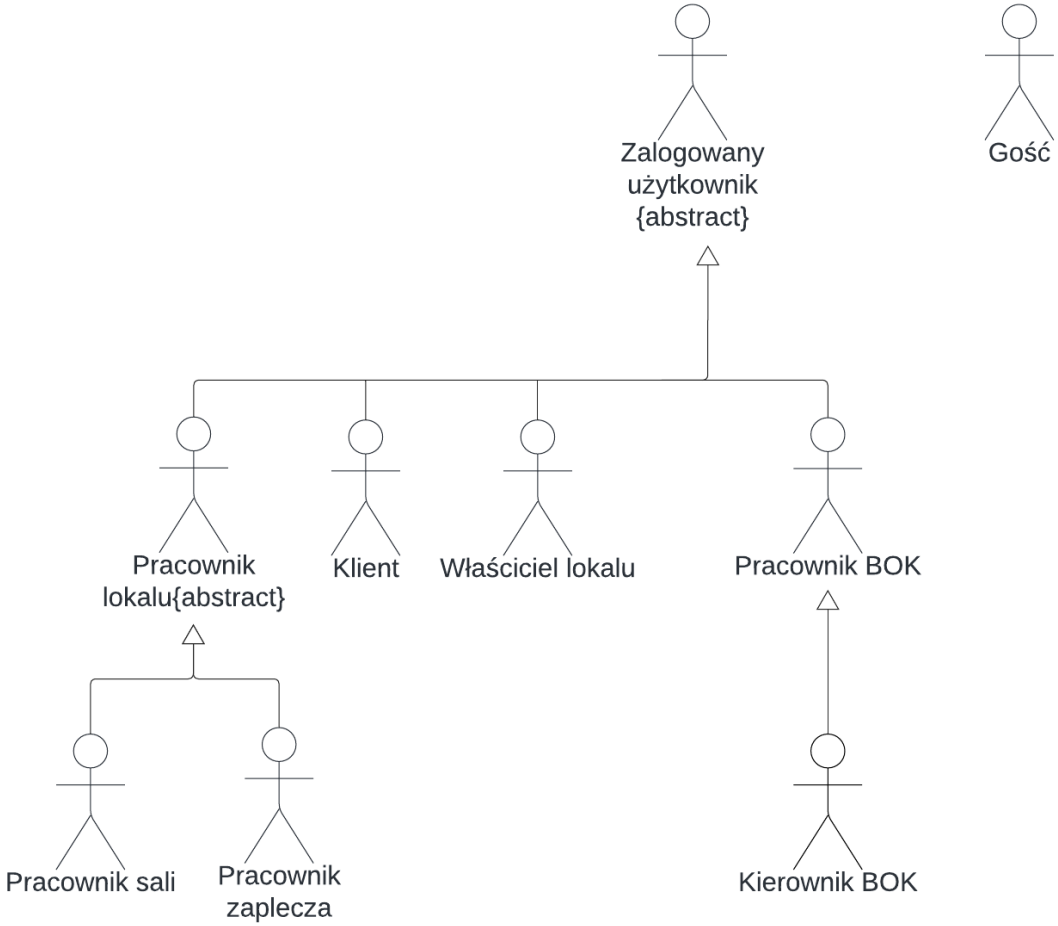
23.	Wyświetlenie stanu magazynowego zaplecza	Użytkownik może wyświetlić stan magazynu dla konkretnego lokalu	Pracownik Zaplecza, Właściciel lokalu, Pracownik BOK, Kierownik BOK	Użytkownik posiada zarejestrowany w systemie lokal
24.	Zmiana statusu zamówienia	Użytkownik może zmienić stan dla zamówienia, wyróżniamy 4 rodzaje statusów dla zamówienia: “w przygotowaniu”, “gotowy do odbioru”, “odebrany”, “anulowany”	Pracownik Zaplecza, Pracownik BOK, Kierownik BOK	Użytkownik jest przypisany do obsługi zamówienia
25.	Zgłoszenie zapotrzebowania na dany składnik i określenia jego tymczasowej niedostępności	Użytkownik może zgłosić zapotrzebowanie na dany składnik- gdy stwierdzi jego brak na stanie lub określić tymczasową niedostępność składnika	Pracownik Zaplecza, Pracownik BOK, Kierownik BOK	Użytkownik posiada uprawnienia do zarządzania stanem magazynu danej restauracji
26.	Zgłoszenie skargi na klienta	Użytkownik może zgłosić skargę na klienta w związku z jego nieodpowiednim zachowaniem podczas pobytu w lokalu	Pracownik Zaplecza, Pracownik Sali, Pracownik BOK, Kierownik BOK	Skargę można złożyć na użytkownika, który posiadał rezerwację w lokalu
27.	Wyświetlenie zamówienia, które obecnie obsługuje	Użytkownik może wyświetlić i zobaczyć szczegóły zamówienia, które obecnie obsługuje	Pracownik Zaplecza, Pracownik Sali, Pracownik BOK, Kierownik BOK	Szczegóły danego zamówienia może zobaczyć pracownik, który obsługuje dane zamówienie
28.	Wyświetlenie listy stolików, wraz z ich statusami	Użytkownik może wyświetlić listę wszystkich stolików znajdujących się w lokalu i dla każdego z nich sprawdzić stan: “zajęty”, “wolny”, “zarezerwowany”	Pracownik Sali, Pracownik BOK, Kierownik BOK	Dla danej restauracji, przynajmniej jeden stolik znajdujący się w lokalu jest zarejestrowany

29.	Obsługa zgłoszonej rezerwacji (akceptacja, odrzucenie, zmiana numeru stolika przypisanego do rezerwacji)	Użytkownik może zaakceptować/ odrzucić rezerwację lub zmienić numer stolika przypisany do danej restauracji	Pracownik Sali, Pracownik BOK, Kierownik BOK	Dla danej restauracji, przynajmniej jedna rezerwacja została zgłoszona
30.	Wyświetlanie zgłoszeń	Użytkownik może wyświetlać zgłoszenia. Przez zgłoszenie rozumiemy- poinformowanie obsługi systemu o wystąpieniu problemu. Każdy aktor ma dostęp do konkretnych kategorii zgłoszenia: **Klient** - zgłoszenie zgubionego przedmiotu, zgłoszenie pracownika lokalu **Pracownik lokalu** - zgłoszenie Klientów uczestniczących w wizycie w lokalu **Zalogowany użytkownik** - zgłoszenie problemów technicznych z aplikacją	Pracownik BOK, Kierownik BOK	Jest zgłoszone minimum jedno zgłoszenie
31.	Rozpatrzenie zgłoszenia	Użytkownik może rozpatrywać zgłoszenia, W przypadku zgłoszenia zgłoszonego przez Pracownika lokalu dotyczącego nieodpowiedniego zachowania Klientów podczas wizyty w lokalu- w razie pozytywnego rozpatrzenia zgłoszenia ocena wszystkich uczestników wizyty zostanie obniżona, a w szczególnych przypadkach ich konta mogą zostać zbanowane czasowo lub bezterminowo)	Pracownik BOK, Kierownik BOK	Jest zgłoszone minimum jedno zgłoszenie
32.	Wyświetlenie informacji o zamówieniu dot. zgłoszenia	Dla zgłoszenia, które jest w trakcie rozpatrywania użytkownik może wyświetlić informacje na temat	Pracownik BOK, Kierownik BOK	Zgłoszenie jest w trakcie rozpatrywania

		rezerwacji oraz zamówienia, którego dotyczy to zgłoszenie		
33.	Wyświetlenie informacji o zalogowanym użytkowniku, który utworzył zgłoszenie	Dla zgłoszenia, które jest w trakcie rozpatrywania użytkownik może wyświetlić informacje dotyczące zalogowanego użytkownika, który utworzył zgłoszenie	Pracownik BOK, Kierownik BOK	Zgłoszenie jest w trakcie rozpatrywania
34.	Wyświetlenie informacji o zalogowanym użytkowniku, którego dotyczy zgłoszenie	Dla zgłoszenia, które jest w trakcie rozpatrywania użytkownik może wyświetlić informacje dotyczące zalogowanego użytkownika, którego dotyczy zgłoszenie	Pracownik BOK, Kierownik BOK	Zgłoszenie jest w trakcie rozpatrywania
35.	Przyjęcie lokalu	Użytkownik może zatwierdzić rejestrację lokalu w aplikacji.	Kierownik BOK	Lokal spełnia normy prawne oraz podane są wszystkie niezbędne informacje o tym lokalu
36.	Zablokowanie/ odblokowanie konta użytkownika na określony czas	Użytkownik może zablokować/ odblokować konto użytkownikowi aplikacji na określony czas	Kierownik BOK	Kierownik BOK musi podać powód zablokowania/ odblokowania użytkownikowi konta

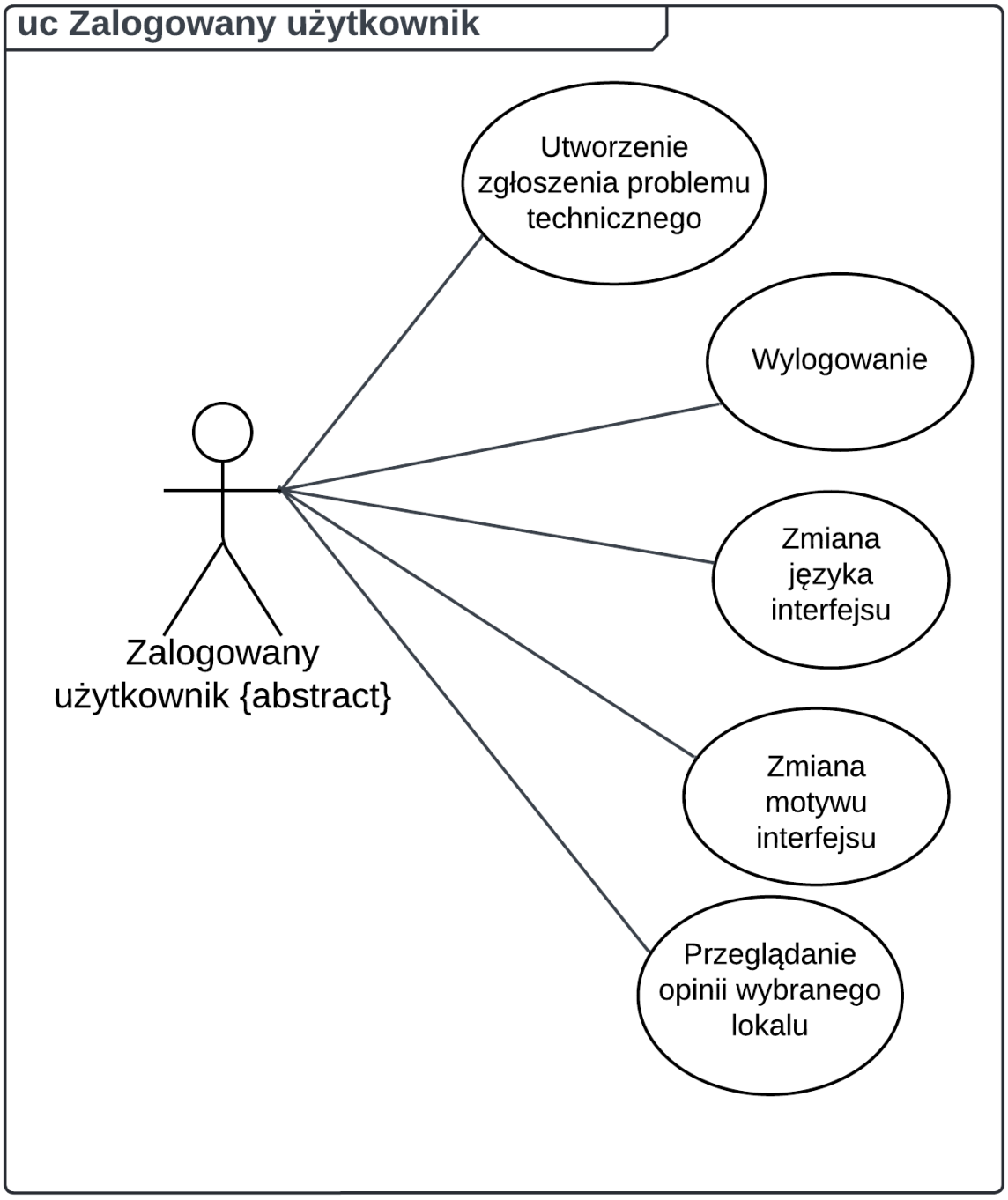
3.6 Aktorzy systemu

Rysunek 21 przedstawia diagram ilustrujący aktorów systemu Reservant:

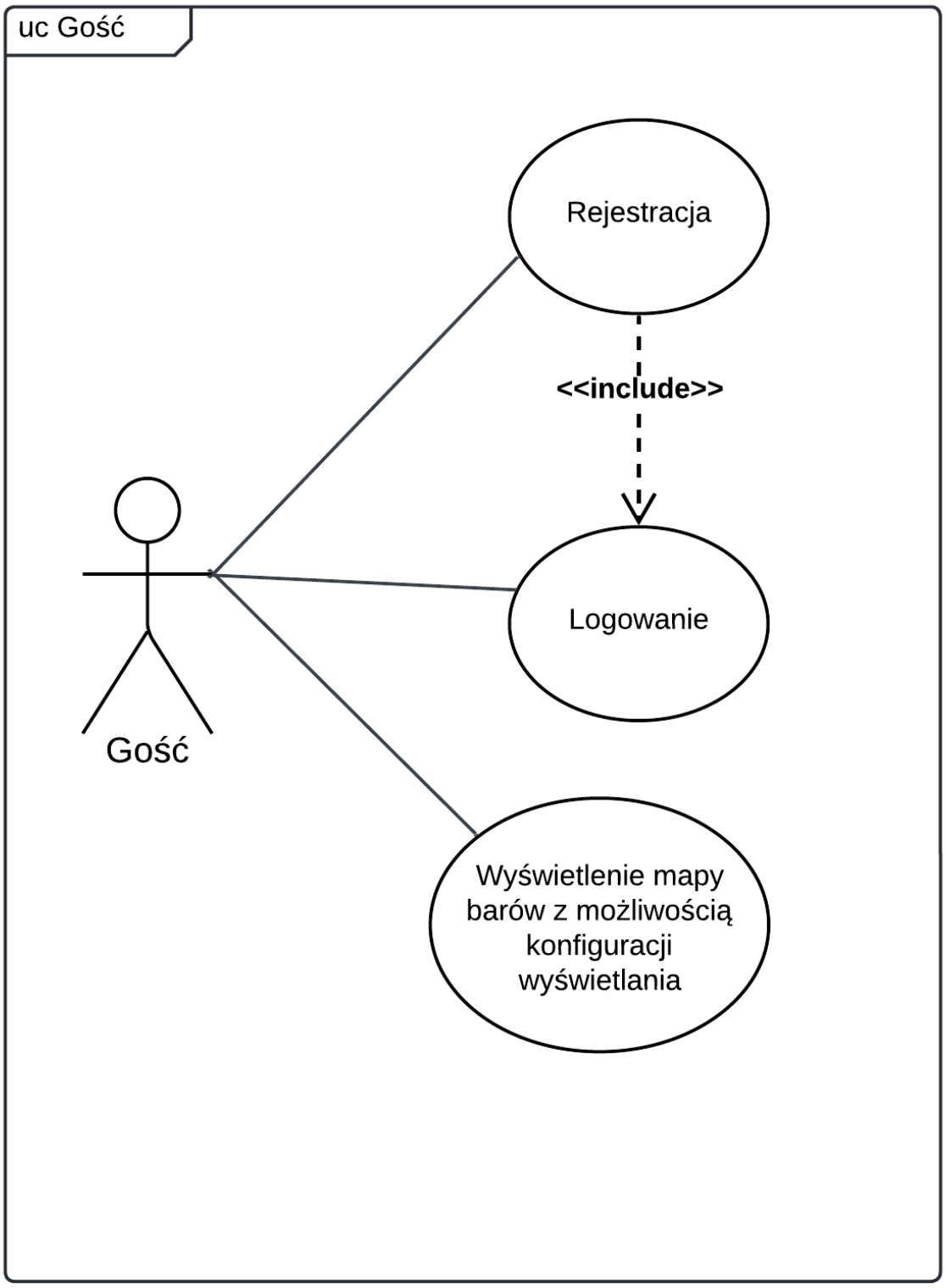


Rysunek 21 Diagram aktorów w systemie Reservant. Źródło: Opracowanie własne

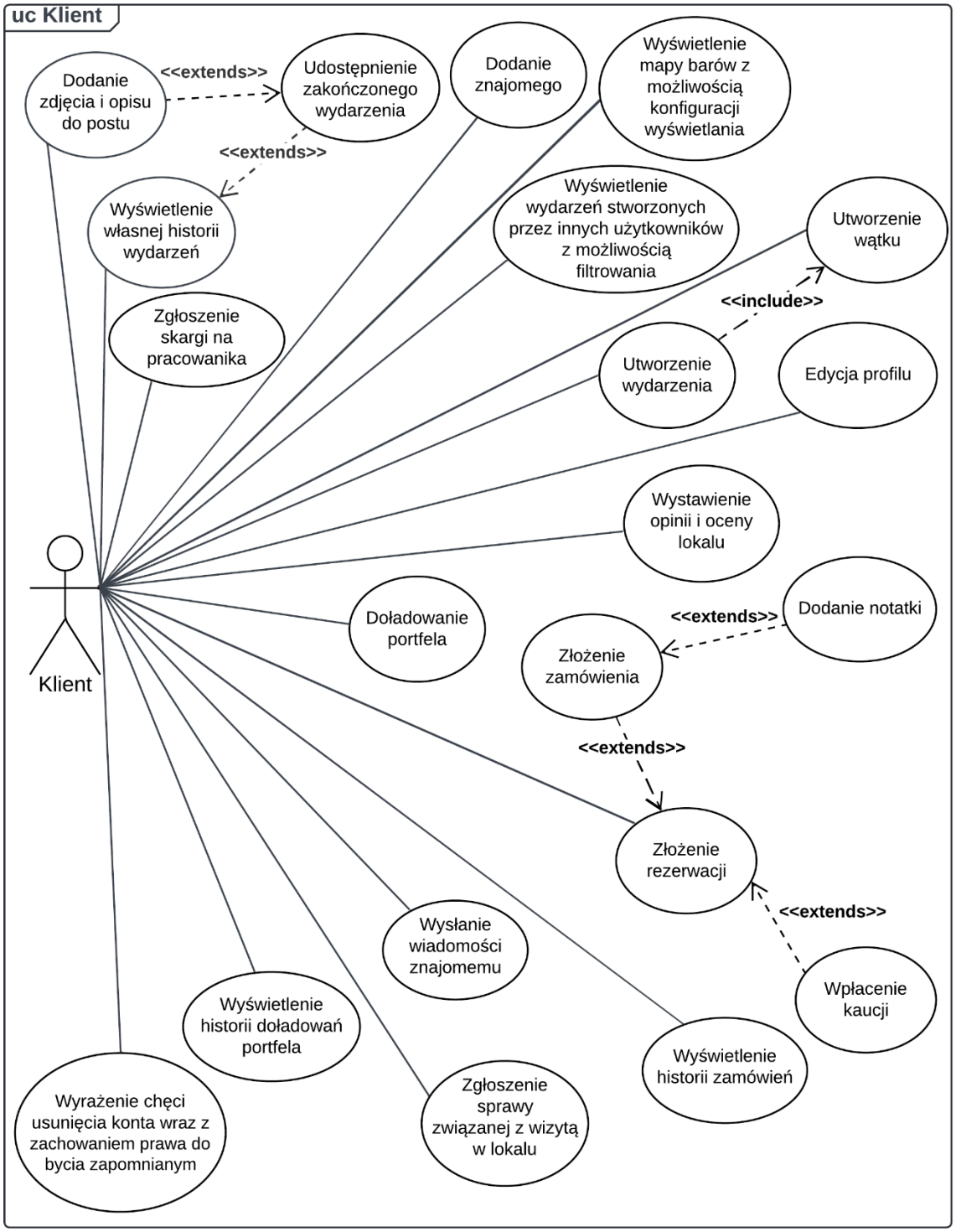
3.7 Przypadki użycia



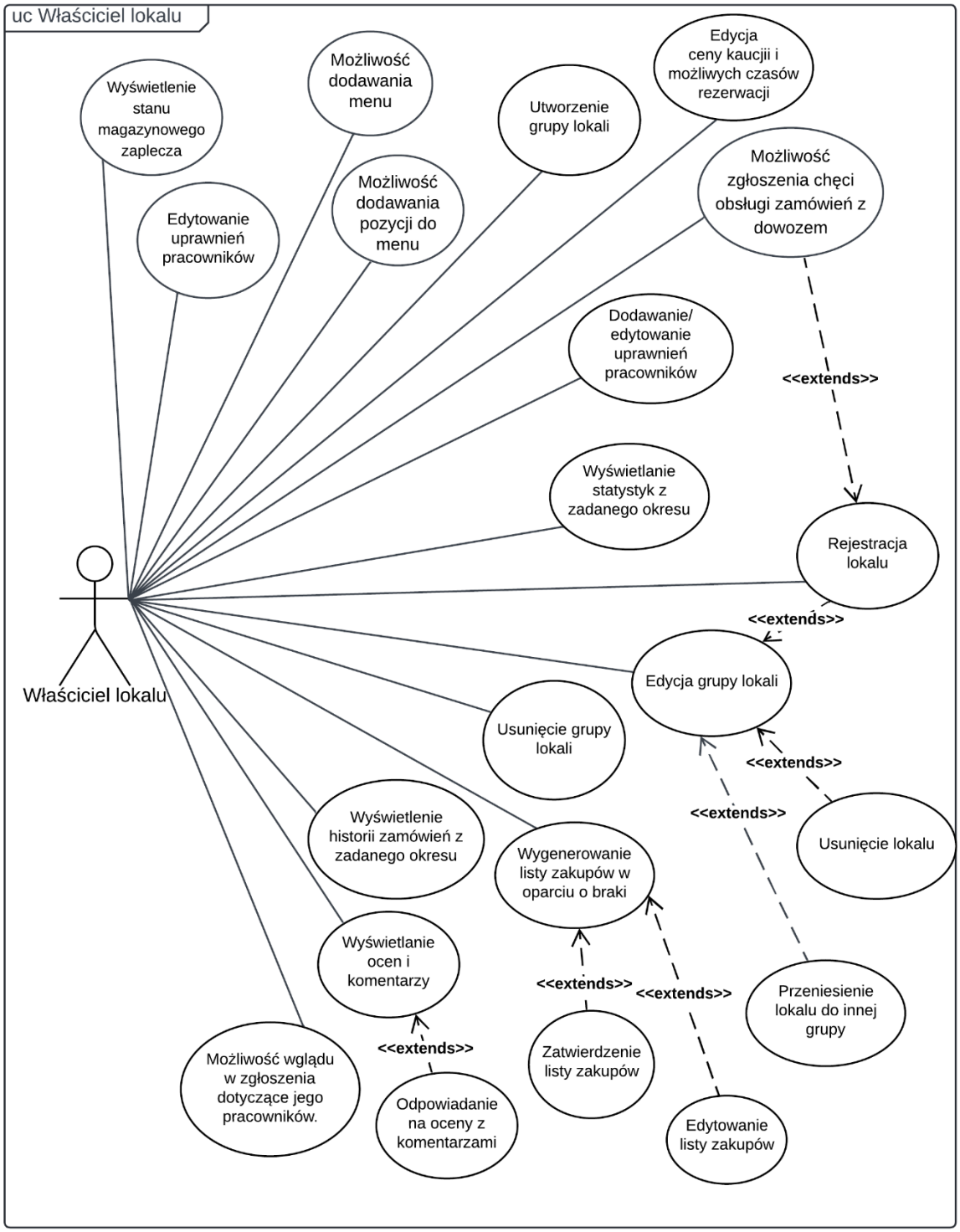
Rysunek 22. Diagram przypadków użycia zalogowanego użytkownika. Źródło: Opracowanie własne



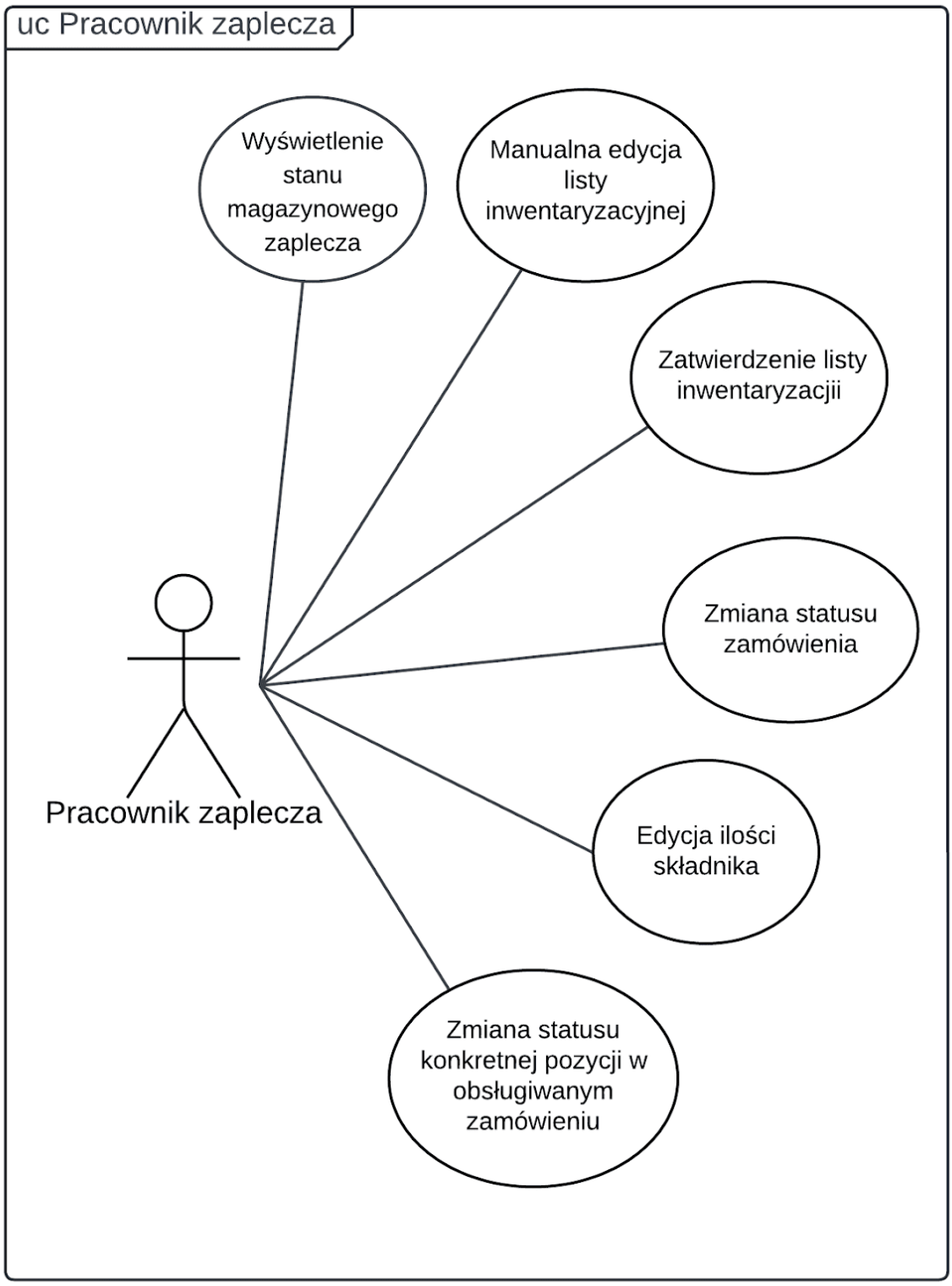
Rysunek 23. Diagram przypadków użycia gościa. Źródło: Opracowanie własne



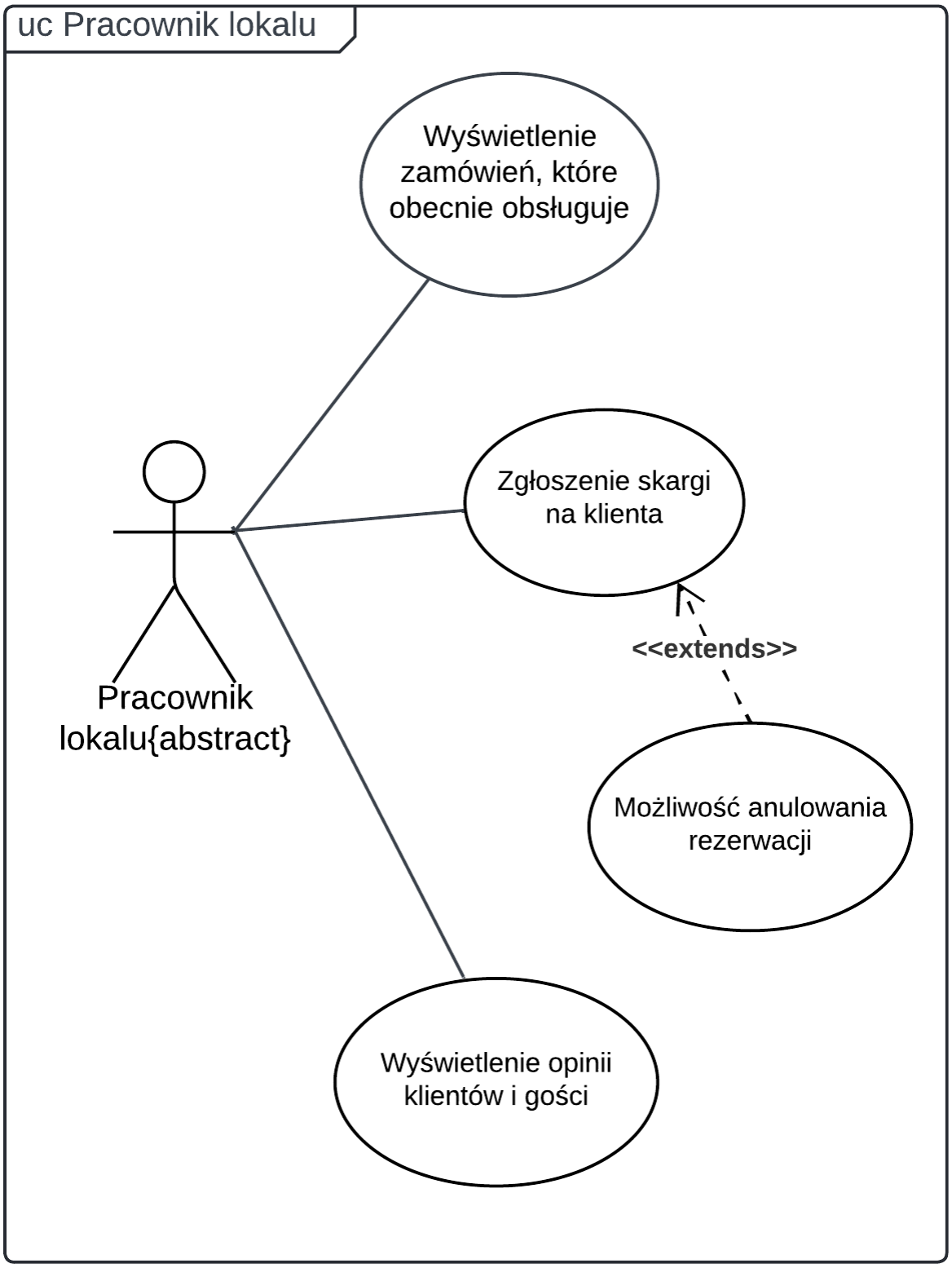
Rysunek 24. Diagram przypadków użycia klienta. Źródło: Opracowanie własne



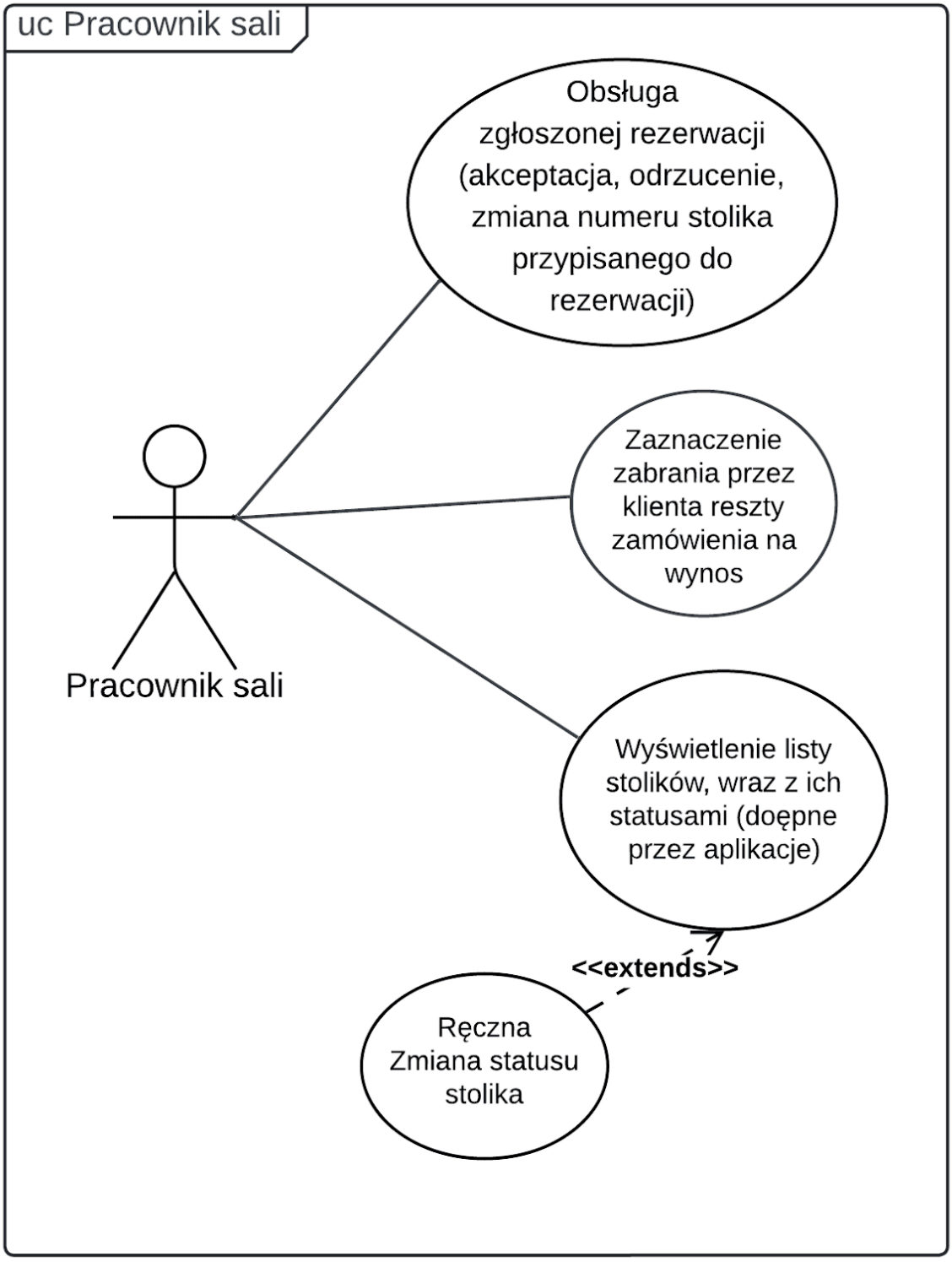
Rysunek 25. Diagram przypadków użycia właściciela lokalu. Źródło: Opracowanie własne



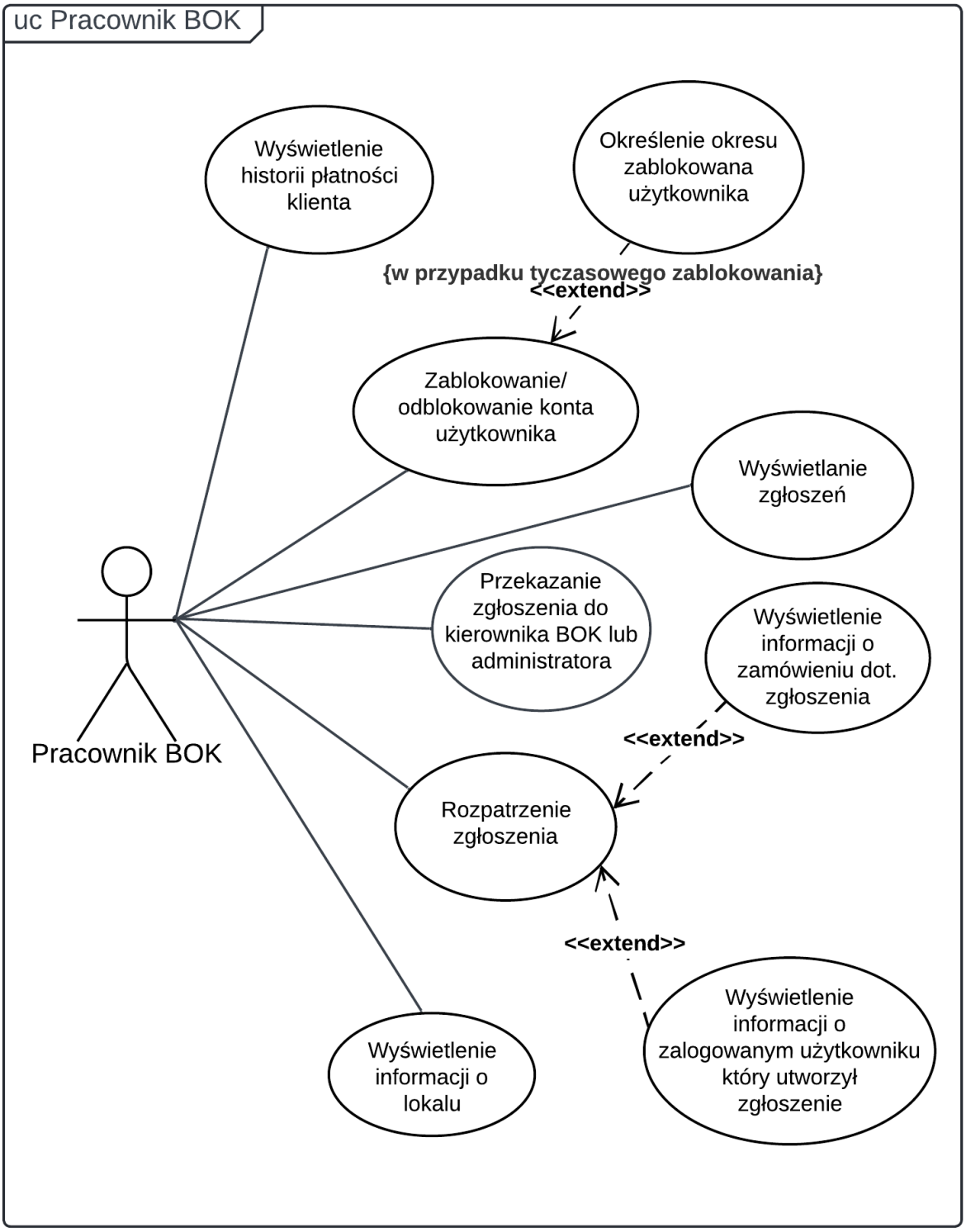
Rysunek 26. Diagram przypadków użycia pracownika zaplecza. Źródło: Opracowanie własne



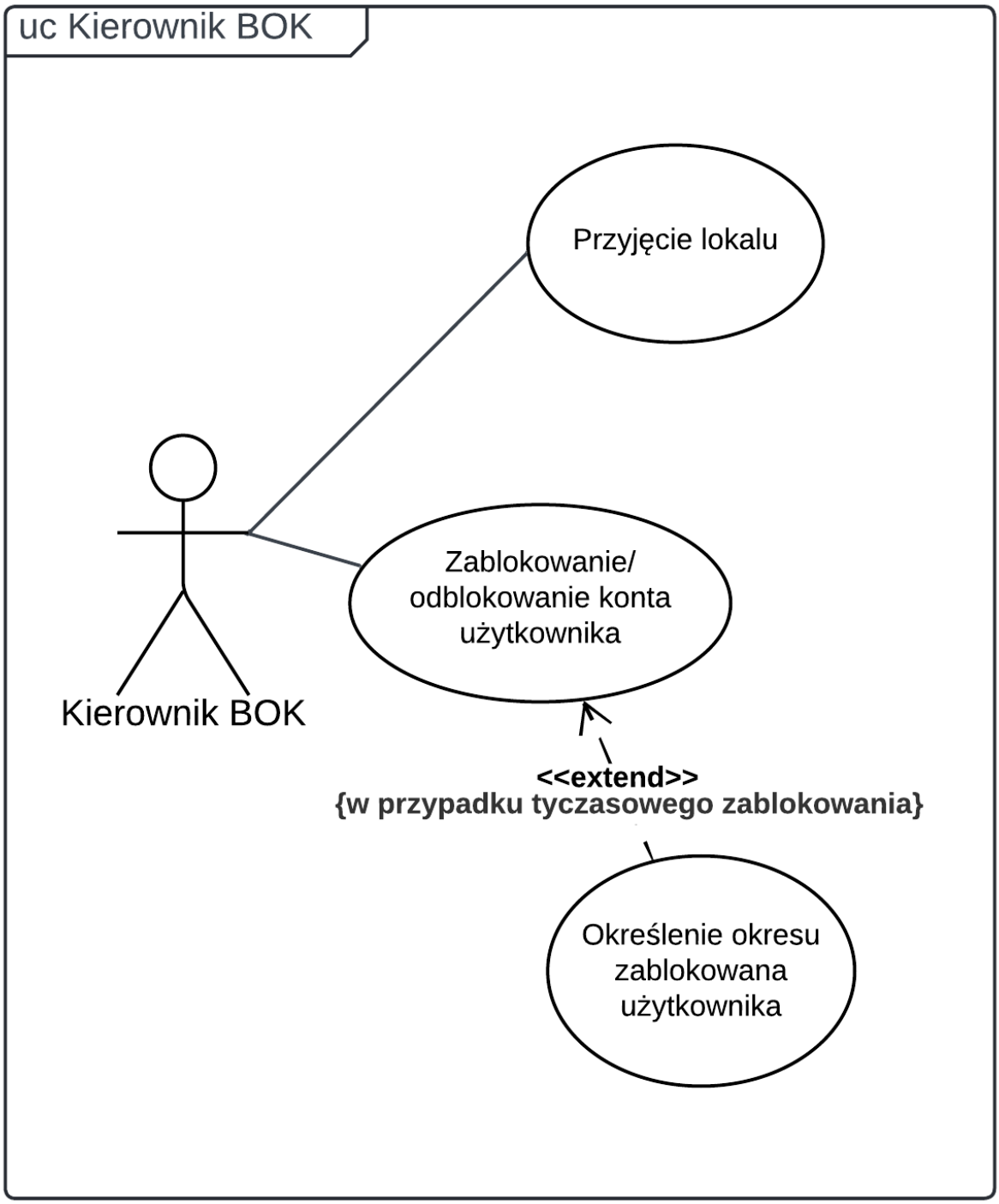
Rysunek 27. Diagram przypadków użycia pracownika lokalu. Źródło: Opracowanie własne



Rysunek 28. Diagram przypadków użycia pracownika Sali. Źródło: Opracowanie własne



Rysunek 29. Diagram przypadków użycia pracownika BOK. Źródło: Opracowanie własne



Rysunek 30. Diagram przypadków użycia Kierownika BOK. Źródło: Opracowanie własne

3.8 Wymagania niefunkcjonalne

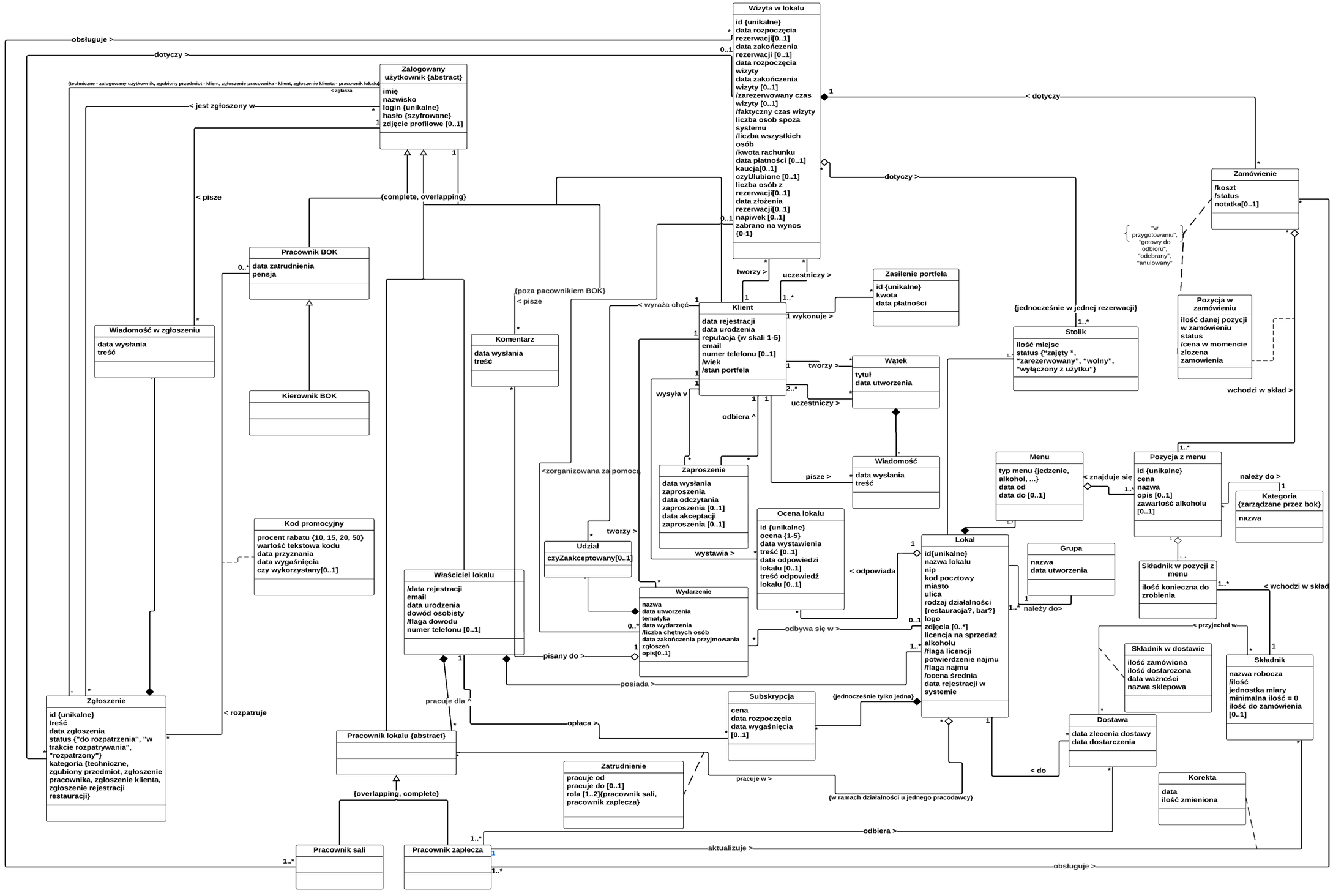
Wymagania niefunkcjonalne systemu to kryteria określające jakościowe aspekty systemu, które definiują, jak system ma działać, ale nie określają konkretnej funkcjonalności. Są kluczowe dla spełnienia oczekiwań użytkowników i zapewnienia odpowiedniego działania systemu w różnych warunkach.

Tabela 12 Wymagania niefunkcjonalne. Źródło: Opracowanie własne

Lp.	Opis	Aspekt
1.	Aplikacja mobilna powinna być responsywna i dynamicznie dostosowywać się do różnych rozmiarów ekranu.	Wydajność
2.	System zapewnia krótki czas odpowiedzi na żądania użytkowników, do 5 sekund.	
3.	Serwer powinien być w stanie obsłużyć ponad 530 zapytań na sekundę	
4.	System powinien zapewniać szyfrowanie haseł za pomocą algorytmu HMAC-SHA512 z liczbą iteracji wynoszącą 100.000	Bezpieczeństwo
5.	System powinien zapewniać kontrolę dostępu do zasobów systemu	
6.	System jest dostępny w języku angielskim i polskim	Użyteczność
7.	Aplikacja webowa systemu powinna być dostępna w przeglądarkach internetowych opartych na silniku chromium	
8.	Aplikacja mobilna powinna być dostępna na urządzeniach mobilnych z systemem Android	
9.	System wymaga 1TB pamięci masowej	Niezawodność
10.	System wymaga 4GB pamięci operacyjnej (RAM)	
11.	System powinien być dostępny w 99% czasu	

3.9 Analityczny diagram klas

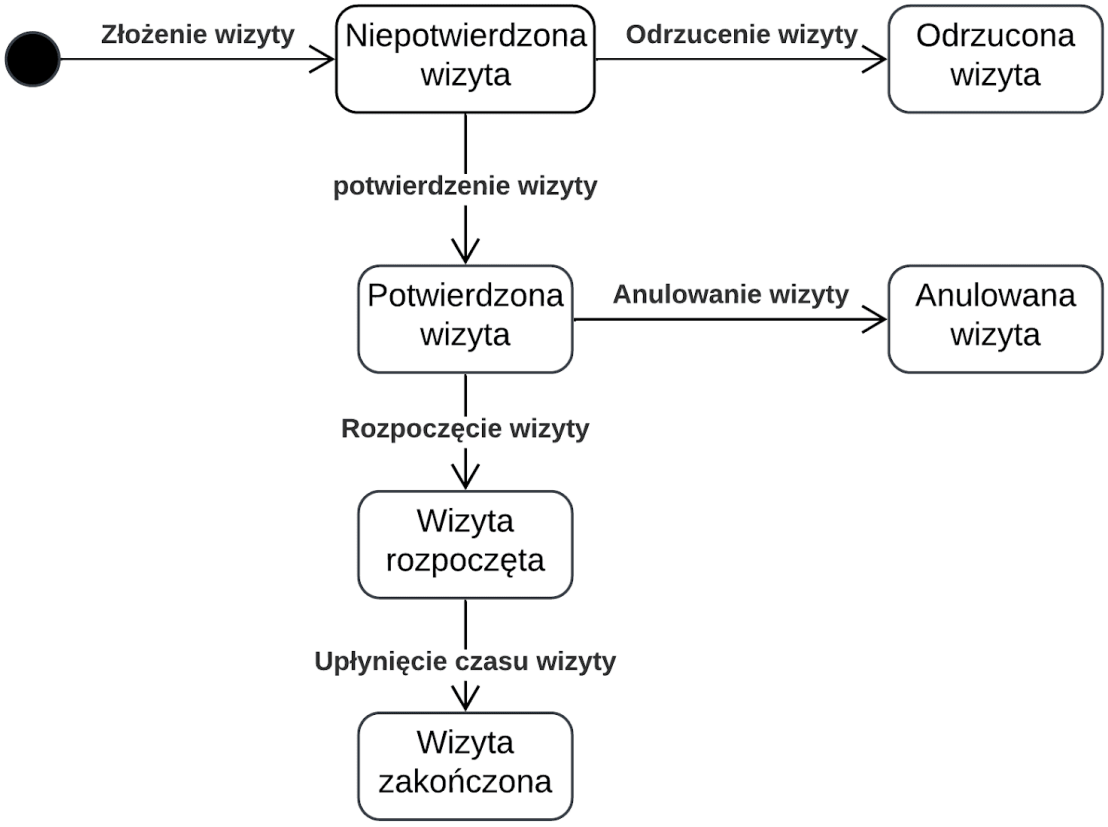
Rysunek 31 przedstawia strukturę systemu do zarządzania lokalem gastronomicznym, obejmującą relacje między głównymi encjami i ich atrybutami. System ten wspiera zarządzanie klientami, pracownikami, rezerwacjami, zamówieniami, produktami oraz zasobami lokalu.



Rysunek 31 Analityczny diagram klas. Źródło: Opracowanie własne

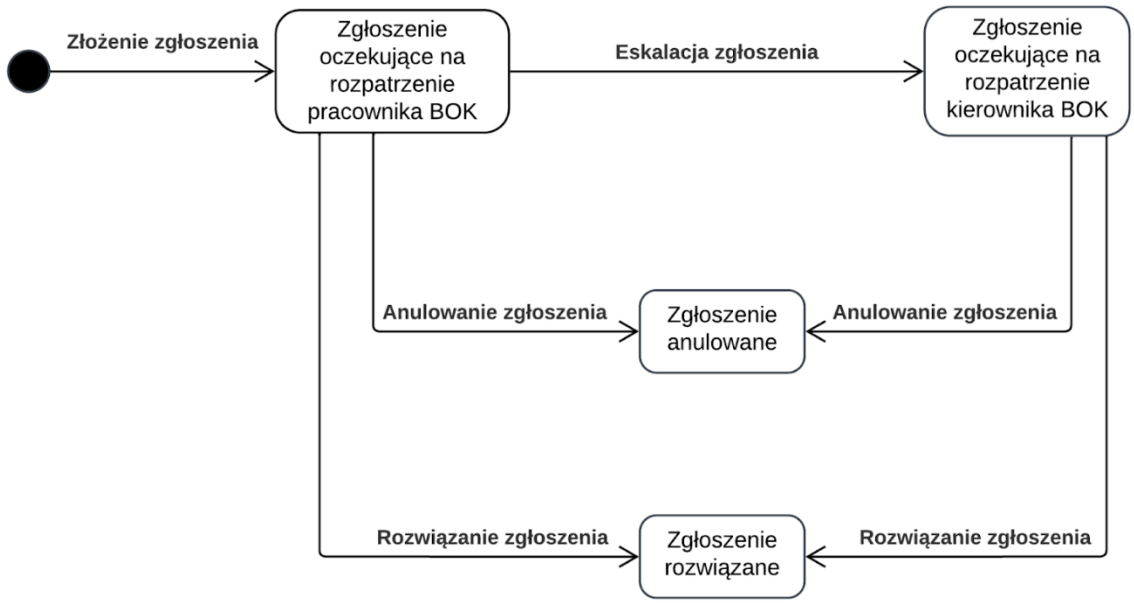
3.10 Diagramy stanów

Rysunek 32 przedstawia diagram stanów złożonej wizyty. Aby wizyta mogła się rozpocząć, system oczekuje na potwierdzenie. W przypadku jego braku, wizyta zostaje odrzucona. Po potwierdzeniu klient może rozpocząć wizytę lub ją anulować



Rysunek 32 Diagram stanów wizyty. Źródło: Opracowanie własne

Rysunek 33 przedstawia diagram stanów złożonego zgłoszenia. Złożone zgłoszenie trafia do biura obsługi klienta, gdzie zależnie od decyzji osoby rozpatrującej zostaje anulowane lub rozwiązane.



Rysunek 33 Diagram stanów zgłoszenia. Źródło: Opracowanie własne.

Rysunek 34 prezentuje diagram stanów dla obiektu restauracji. W pierwszej kolejności restauracja musi zostać zweryfikowana przez pracowników BOK pod kątem poprawności danych oraz nadesłanych dokumentów. Po udanej weryfikacji restauracja staje się publicznie dostępna.

Rysunek 34 Diagram stanów restauracji. Źródło: Opracowanie własne.

3.11 Scenariusze

W tym rozdziale można zapoznać się ze scenariuszami dla najważniejszych przypadków użycia powstałych podczas fazy analitycznej.

Tabela 13 Scenariusz przypływu zdarzeń przypadku „Złożenie rezerwacji". Źródło: Opracowanie własne

Nazwa	Złożenie rezerwacji	
Aktorzy:	Klient, Pracownik sali	
Główny przepływ zdarzeń:		
L.P.	Nazwa aktywności	Numer następnej aktywności:
1	Klient uruchamia aplikację i loguje się	2
2	Klient wybiera restaurację, w której chce złożyć rezerwację	3
3	Klient wybiera opcję „Złóż rezerwację"	4
4	Klient wysyła prośbę o rezerwację stolika w lokalu na ten sam dzień	5
5	Pracownik Sali akceptuje rezerwację	6
6	Klient otrzymuje informację o akceptacji rezerwacji	7
7	Pracownik lokalu potwierdza rezerwację	8
8	Klient otrzymuje informację o numerze stolika	9
9	Zakończenie przypadku użycia	
Alternatywny przepływ zdarzeń:		
L.P.	Nazwa aktywności	Numer następnej aktywności:
4 a	Klient wysyła prośbę o rezerwację stolika w lokalu na inny dzień	4.1 a
4.1 a	Klient otrzymuje informację o automatycznej akceptacji rezerwacji	7
5 a	Pracownik Sali odrzuca rezerwację	5.1 a
5.1 a	Klient otrzymuje informację o odrzuceniu rezerwacji	9

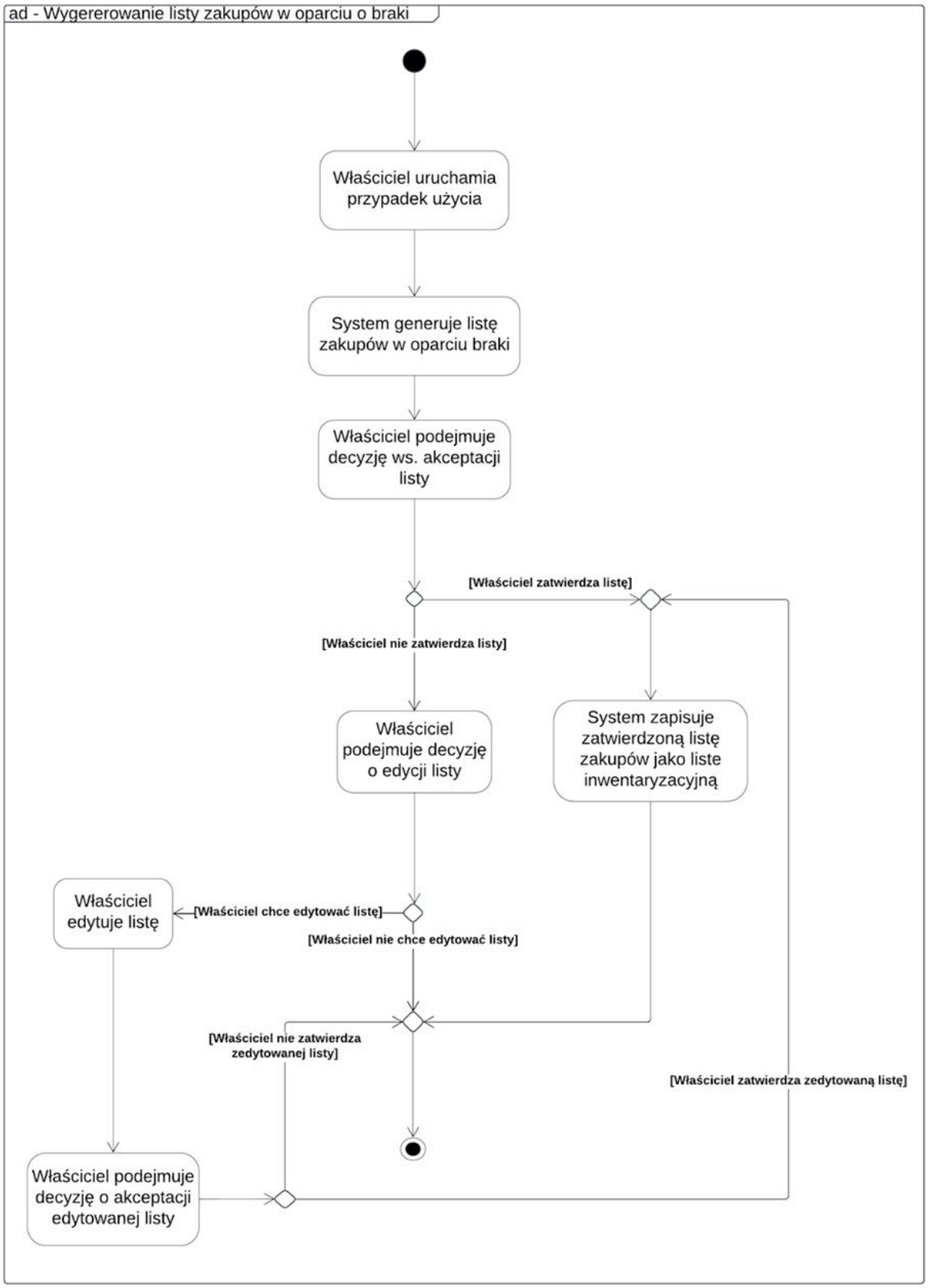
Tabela 14 Scenariusz przypływu zdarzeń przypadku „Udział w wydarzeniu”. Źródło: Opracowanie własne

Nazwa	Udziału w wydarzeniu	
Aktorzy:	Klient	
Główny przepływ zdarzeń:		
L.P.	Nazwa aktywności	Numer następnej aktywności:
1	Klient uruchamia aplikację i loguje się	2
2	Klient wchodzi na stronę główną	3
3	Klient wybiera wydarzenie z listy wydarzeń	4
4	Klient wybiera opcję „Weź udział w wydarzeniu”	5
5	Klient otrzymuje potwierdzenie wzięcia udziału w wydarzeniu	6
6	Zakończenie przypadku użycia	
Alternatywny przepływ zdarzeń:		
L.P.	Nazwa aktywności	Numer następnej aktywności:
2 a	Klient wybiera restaurację	3
2 b	Klient wybiera restaurację	2.1 b
2.1 b	Klient wybiera opcję „Utwórz wydarzenie”	2.2 b
2.2 b	Klient wprowadza dane dla wydarzenia	2.3 b
2.3 b	Klient otrzymuje potwierdzenie utworzenia wydarzenia	3

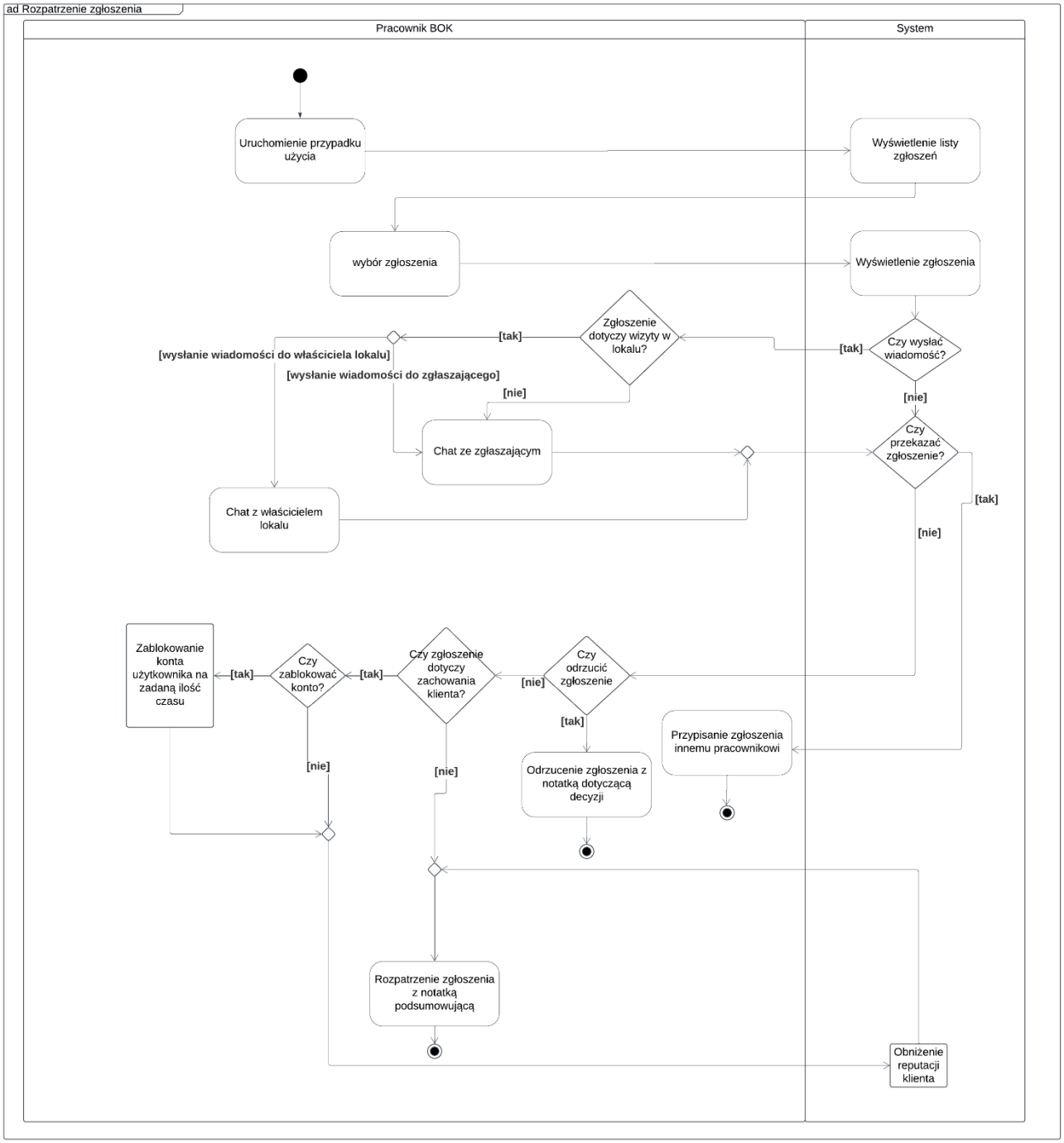
Tabela 15 Scenariusz przypływu zdarzeń przypadku „Rejestracja lokalu”. Źródło: Opracowanie własne

Nazwa	Rejestracja lokalu	
Aktorzy:	Właściciel lokalu	
Główny przepływ zdarzeń:		
L.P.	Nazwa aktywności	Numer następnej aktywności:
1	Właściciel lokalu uruchamia aplikację i loguje się	2
2	Właściciel lokalu wchodzi na stronę główną	3
3	Właściciel lokalu wybiera zakładkę „Moje restauracje”	4
4	Właściciel lokalu wybiera opcję „Dodaj restaurację”	5
5	Właściciel lokalu wypełnia formularz dodania nowej restauracji	6
6	Właściciel lokalu otrzymuje potwierdzenie dodania nowej restauracji	7
7	Zakończenie przypadku użycia	

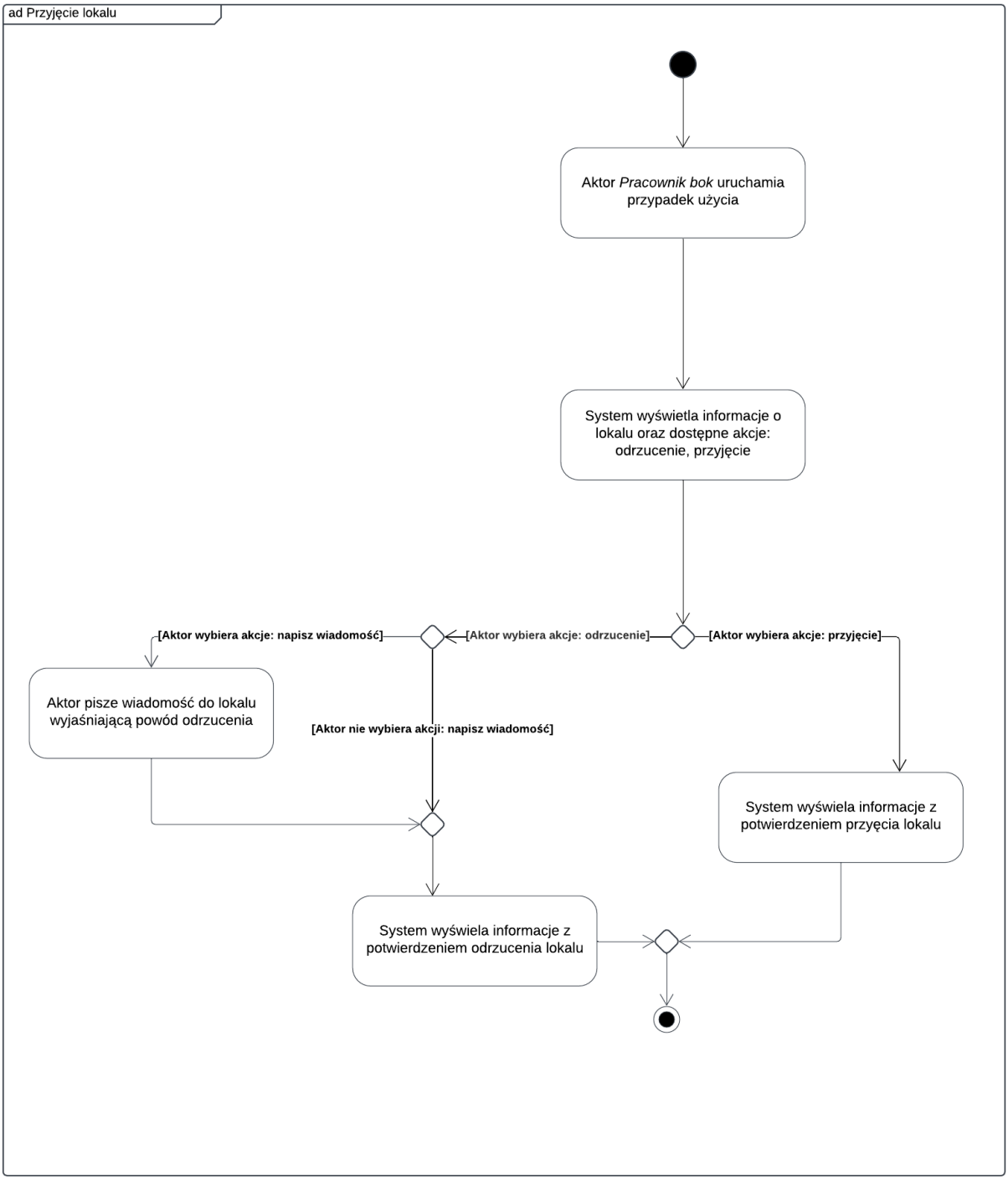
3.12 Diagramy aktywności



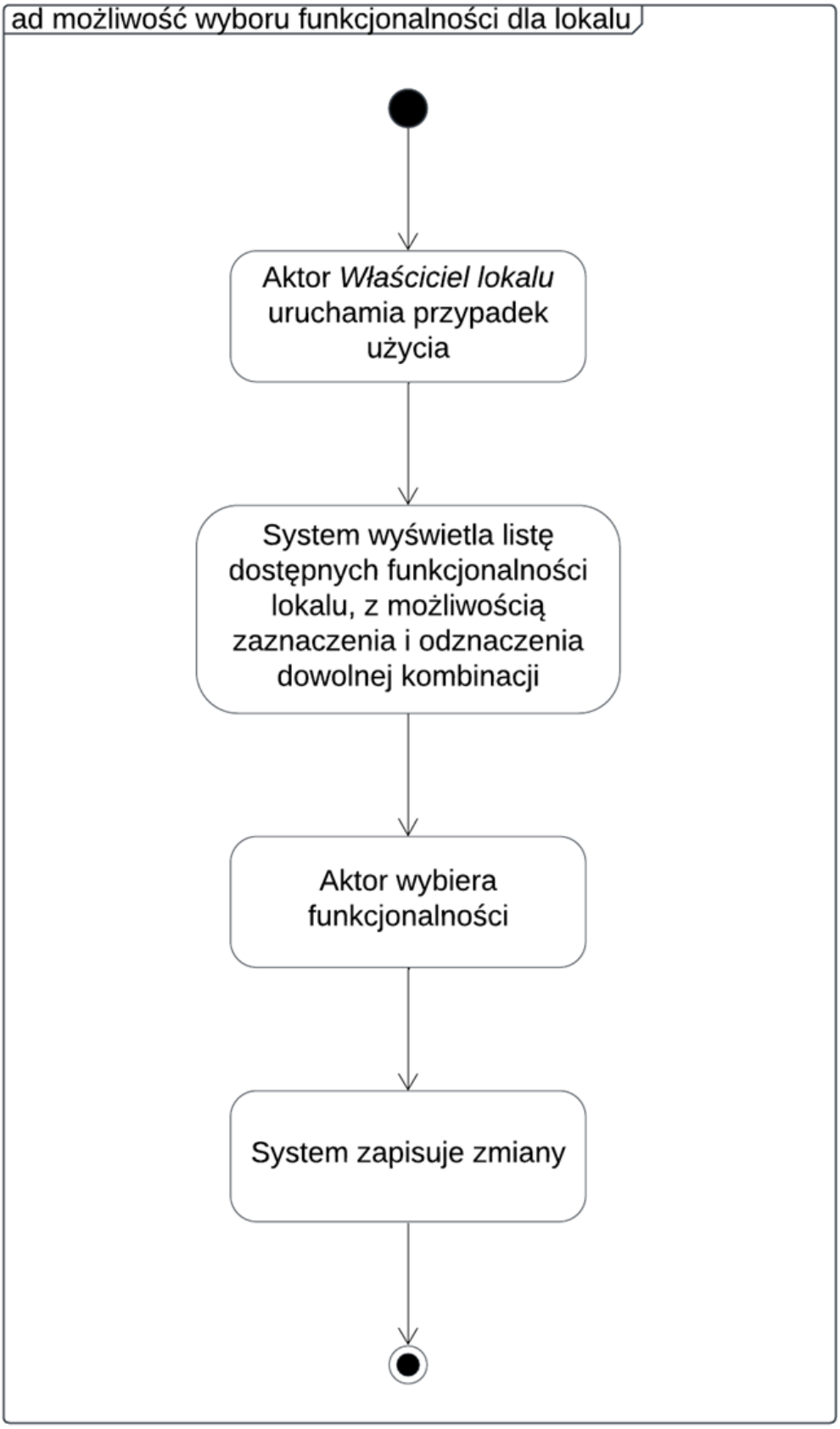
Rysunek 35 Wygenerowanie listy zakupów. Źródło: Opracowanie własne



Rysunek 36 Rozpatrzenie zgłoszenia przez pracownika BOK. Źródło: Opracowanie własne



Rysunek 37 Proces akceptacji lokalu. Źródło: Opracowanie własne



Rysunek 38 Możliwość wyboru funkcjonalności. Źródło: Opracowanie własne

4 Technologie oraz narzędzia

Rozdział ten opisuje technologię i narzędzia używane podczas etapu realizacji projektu. Celem jest jak najdokładniejsze opisanie środowisk informatycznych które pomogły w tworzeniu naszego oprogramowania. Poniżej opisane zostały wszelkie programy, komponenty bądź platformy, które wykorzystane zostały w poszczególnych etapach pracy nad projektem.

4.1 Język programowania C#

C# według [20] jest obiektowym językiem programowania, który został stworzony przez firmę Microsoft jako część platformy .NET. Jest to silnie typowany i kompilowany język, który jest szeroko wykorzystywany do tworzenia różnorodnych aplikacji, takich jak aplikacje internetowe, gry, narzędzia programistyczne i wiele innych.

Jest to język wysokiego poziomu, co oznacza, że oferuje wygodne abstrakcje programistyczne, ułatwiające tworzenie skomplikowanych systemów. Działania odnoszą się do operacji wykonywanych na danych przy użyciu różnych operatorów oraz funkcji wbudowanych w język.

4.1.1 Historia

C# został zaprojektowany przez Andersa Hejlsberga i jego zespół w Microsoft. Pierwsza publiczna wersja została wydana w 2002 roku jako część platformy .NET Framework 1.0. Język ten był odpowiedzią na potrzebę bardziej nowoczesnego i bezpiecznego języka programowania, który byłby łatwy w użyciu dla programistów, a jednocześnie oferowałby zaawansowane funkcje.

Od tego czasu C# ewoluował i był regularnie aktualizowany. Microsoft wydał kilka wersji języka, dodając nowe funkcje, ulepszenia i usprawnienia. Niektóre z najbardziej znaczących wersji to C# 2.0 (2005), C# 3.0 (2007), C# 5.0 (2012), C# 6.0 (2015), C# 7.0 (2017), C# 8.0 (2019) i C# 9.0 (2020).

4.1.2 Składnia

- Zmienne i Typy: W C# zmienne muszą być zadeklarowane z określonym typem przed użyciem. Typy mogą być proste lub złożone;
- Instrukcje sterujące: C# oferuje standardowe instrukcje sterujące takie jak if, else, switch oraz pętle;
- Funkcje i metody: Funkcje są podstawowymi jednostkami strukturalnymi w C#. Po deklaracji mogą zwracać wartość lub jej nie zwracać;
- Klasy i obiekty: C# jest językiem obiektowym, więc opiera się na koncepcji klas i obiektów. Klasy definiują właściwości i metody, podczas gdy obiekty są instancjami tych klas;
- Interfejsy: C# wspiera interfejsy, które oferują między innymi polimorfizm i abstrakcję;
- Kolekcje: C# oferuje różne typy kolekcji, takie jak listy, słowniki, kolejki. Ułatwiają one przechowywanie i zarządzanie danymi.

Listing 1 ilustruje prosty program w C#, który demonstruje deklarację zmiennych, użycie instrukcji warunkowych oraz operacje wejścia-wyjścia

Listing 1 Przykład prostego programu w C#. Źródło: Opracowanie własne.

```
using System;

class Program
```

```
{
    static void Main(string[] args)
    {
        string name = "Alice";
        int age = 30;

        Console.WriteLine($"Hello, my name is {name} and I am {age} years old.");

        if (age >= 18)
        {
            Console.WriteLine("I am an adult.");
        }
        else
        {
            Console.WriteLine("I am a minor.");
        }
    }
}
```

4.1.3 Semantyka

- Silna typizacja: W C# zmienne muszą być zadeklarowane z konkretnym typem i typy muszą być zgodne, zapewnia to bezpieczeństwo typów podczas kompilacji;
- Zarządzanie pamięcią: W języku C# programista nie musi ręcznie zarządzać pamięcią, co zmniejsza ryzyko wycieków pamięci;
- Wielowątkowość: C# wspiera wielowątkowość, co pozwala na równoległe wykonywanie operacji, co jest szczególnie przydatne w aplikacjach, które wymagają równoległego przetwarzania;
- Wyjątki: C# obsługuje wyjątki, co pozwala na zarządzanie błędami i nieprzewidzianymi sytuacjami w trakcie wykonywania programu.

Listing 2 ilustruje przykład wykorzystania asynchroniczności w C#, gdzie zadania wykonywane są równolegle, a zarządzanie zasobami odbywa się w sposób bezpieczny dzięki implementacji interfejsu `IDisposable`.

Listing 2 Przykład asynchronicznego wykonywania zadań. Źródło: Opracowanie własne

```
using System;
using System.Threading.Tasks;
class Resource : IDisposable
{
    public void DoWork()
    {
        Console.WriteLine("Performing work...");
    }

    public void Dispose()
    {
        Console.WriteLine("Releasing resources...");
    }}
class Program
{
    static async Task Main(string[] args)
    {
        using (var resource = new Resource())
        {
            resource.DoWork();
            await PerformAsyncWork();
        }
```

```
    }
    static async Task PerformAsyncWork()
    {
        await Task.Delay(1000);  // simulating asynchronous work
        Console.WriteLine("Asynchronous work completed.");
    }
}
```

4.2 ASP .NET

ASP.NET zgodnie z [22] to kompleksowy framework programistyczny stworzony przez firmę Microsoft. Jest to narzędzie niezwykle wszechstronne, dedykowane tworzeniu zaawansowanych aplikacji internetowych oraz usług sieciowych.

Listing 3 ilustruje przykład podstawowego kontrolera w ASP.NET, który obsługuje zapytania HTTP GET i zwraca prostą odpowiedź "*Hello, World!*"

Listing 3 Przykład podstawowego kontrolera w ASP .NET. Źródło: [21]

```
using Microsoft.AspNetCore.Mvc;

namespace MyWebApp.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class HelloWorldController : ControllerBase
    {
        [HttpGet]
        public ActionResult<string> Get()
        {
            return "Hello, World!";
        }
    }
}
```

Jedną z kluczowych cech ASP.NET jest możliwość programowania w różnych językach, takich jak przykładowo C#. Dzięki temu deweloperzy mogą korzystać z języka, który najlepiej odpowiada wymaganiom projektu i ich własnym preferencjom.

ASP.NET zapewnia także szeroki zakres mechanizmów bezpieczeństwa, w tym autoryzację, uwierzytelnianie, kontrolę dostępu oraz ochronę przed atakami typu XSS (Cross-Site Scripting) czy CSRF (Cross-Site Request Forgery). Integracja z bazami danych, takimi jak SQL Server, MySQL czy PostgreSQL, jest również integralną częścią ASP.NET, dzięki czemu deweloperzy mogą łatwo komunikować się z danymi.

Narzędzia deweloperskie dostarczone przez Microsoft, w tym popularne środowisko programistyczne Visual Studio, znacznie ułatwiają proces tworzenia, debugowania i wdrażania aplikacji ASP.NET. Dzięki nim, programiści mogą efektywnie wykorzystywać potencjał frameworka i tworzyć wydajne, bezpieczne oraz skalowalne aplikacje internetowe.

Struktura projektu ASP.NET może się różnić w zależności od używanej wersji platformy oraz preferencji dewelopera. Zazwyczaj projekt ASP.NET ma następującą strukturę:

- **Folder App_Data**: Podobnie jak w typowym projekcie ASP.NET, ten folder może zawierać pliki danych aplikacji takie jak bazy danych w formacie .mdf lub inne pliki potrzebne do przechowywania danych aplikacji;

- **Folder App_Start**: Tutaj znajdują się pliki konfiguracyjne, które mogą być niezbędne do uruchomienia aplikacji. Może to obejmować ustawienia routingu, konfigurację usług, konfigurację wstrzykiwania zależności i inne;
- **Folder Content**: Ten folder zawiera zasoby statyczne używane w aplikacji takie jak arkusze stylów CSS, obrazy, pliki JavaScript;
- **Folder Controllers**: Tutaj znajdują się kontrolery aplikacji, które obsługują żądania HTTP i decydują które widoki powinny zostać wyświetlone na ekranie;
- **Folder Models**: W tym folderze znajdują się modele danych używane przez aplikację. Modele te reprezentują strukturę danych przechowywanych w bazie danych lub używanych w aplikacji;
- **Folder Scripts**: Ten folder zawiera pliki JavaScript używane w aplikacji;
- **Folder Views**: Tutaj znajdują się pliki widoków aplikacji, są to pliki. cshtml (Technologia Razor) lub .aspx (Technologia Web Forms). Widoki są odpowiedzialne za prezentację danych użytkownikowi;
- **Folder App_GlobalResources**: W przypadku aplikacji wielojęzycznych ten folder może zawierać pliki zasobów takie jak teksty które mogą być tłumaczone na różne języki;
- **Folder App_LocalResources**: Tutaj znajdują się zasoby specyficzne dla pojedynczych stron lub kontrolerów które są używane do tłumaczenia tekstu na stronach lub kontrolerach;
- **Pliki konfiguracyjne**: W katalogu głównym projektu znajdują się pliki konfiguracyjne, takie jak Web.config które zawierają ustawienia konfiguracyjne dla aplikacji, takie jak połączenia do baz danych, ustawienia routingu, ustawienia bezpieczeństwa;

Struktura projektu ASP.NET, opisująca poszczególne foldery i pliki, pomaga w organizacji aplikacji w sposób czytelny i efektywny. Listing 4 pokazuje przykład implementacji struktury projektu ASP.NET, w której zarejestrowano kontrolery, skonfigurowano *middleware* oraz zdefiniowano podstawowy kontroler obsługujący zapytania HTTP GET.

Listing 4 Przykład struktury projektu ASP .NET. Źródło: [21]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers(); // Register services

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
      app.UseDeveloperExceptionPage();
}
else
{
      app.UseExceptionHandler("/Home/Error");
      app.UseHsts();
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers(); // Map attribute routing

app.Run(); // Run the application
```

```
[ApiController]
[Route("api/[controller]")]
public class HelloWorldController : ControllerBase
{
      [HttpGet]
      public ActionResult<string> Get() => "Hello, World!";
}
```

4.3 Entity Framework Core

Według [24] EF Core jest lekką, modularną i cross-platformową wersją Entity Framework, przeznaczoną dla .NET Core i .NET 5+. Jest to narzędzie umożliwiające mapowanie obiektowo-relacyjne, co oznacza, że pozwala programistom modelować dane bazodanowe za pomocą klas i relacji obiektów, zamiast operować bezpośrednio na tabelach i kolumnach w bazie danych. EF Core dostarcza zestaw narzędzi do tworzenia, modyfikowania, usuwania i pobierania danych z bazy danych za pomocą standardowych konstrukcji języka programowania, co ułatwia zarządzanie danymi i ich manipulację.

Listing 5 przedstawia przykład podstawowej konfiguracji EF Core, w którym zdefiniowano kontekst bazy danych, tabele odpowiadające encjom oraz relację jeden-do-wielu między autorami a książkami.

Listing 5 Przykład podstawowej konfiguracji Entity Framework. Źródło: [23]

```
using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext
{
    public DbSet<Author> Authors { get; set; }
    public DbSet<Book> Books { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {

optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=LibraryDb;Trusted_Conn
ection=True;");
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Author>()
            .HasMany(a => a.Books)
            .WithOne(b => b.Author)
            .HasForeignKey(b => b.AuthorId);
    }
}
```

4.3.1 Modele danych

Modele danych w EF Core są reprezentowane jako klasy POCO (ang. *Plain Old CLR Objects*), czyli zwykłe obiekty języka C#, które nie wymagają specjalnych atrybutów ani implementacji interfejsów. Klasy te mapowane są na odpowiednie tabele w bazie danych. Każde pole w klasie odpowiada kolumnie w tabeli, a relacje między klasami są odwzorowywane na relacje między tabelami. Aby zdefiniować model danych w EF Core, programista tworzy klasy reprezentujące encje i określa relacje między nimi.

4.3.2 Migracje

Migracje w EF Core są mechanizmem umożliwiającym zarządzanie zmianami struktury bazy danych w sposób kontrolowany i przewidywalny. Pozwalają one na automatyczne generowanie skryptów SQL, które wprowadzają niezbędne zmiany do bazy danych w celu dostosowania jej do zmian w modelu danych. Proces tworzenia migracji polega na definiowaniu zmian w modelu danych za pomocą specjalnych poleceń w konsoli kreatora migracji, a następnie generowaniu i aplikowaniu migracji na bazie danych. EF Core śledzi historię migracji, co pozwala na łatwe cofanie i ponowne stosowanie zmian w strukturze bazy danych.

4.3.3 Relacje między tabelami

W EF Core relacje między tabelami są definiowane za pomocą właściwości nawigacyjnych w klasach modelu danych. Istnieją trzy główne rodzaje relacji: jeden do jeden, jeden do wielu i wiele do wielu. Relacje te są mapowane na klucze obce i klucze główne w bazie danych. EF Core automatycznie tworzy odpowiednie klucze obce i zarządza relacjami między tabelami.

4.4 Fluent Validation

Według [25] FluentValidation to biblioteka walidacji danych dla platformy .NET. Pozwala ona programistom definiować reguły walidacji w czytelny sposób, co ułatwia zarządzanie i utrzymanie kodu walidacyjnego. Technologia ta jest popularna w ekosystemie .NET ze względu na swoją elastyczność i wygodę użycia.

4.4.1 Kluczowe cechy FluentValidation:

- **Deklaratywna składnia**: FluentValidation umożliwia definiowanie reguł walidacji w sposób deklaratywny, co oznacza, że programiści mogą czytać i zrozumieć te reguły bez konieczności analizowania kodu w poszukiwaniu warunków walidacji,
- **Podział logiki**: Pozwala oddzielić logikę walidacji od logiki biznesowej, prowadzi to do bardziej czytelnego i modułowego kodu,
- **Recykling**: Reguły walidacji mogą być ponownie wykorzystywane w różnych częściach aplikacji, przyspiesza to proces tworzenia i utrzymania aplikacji,
- **Wsparcie złożonych walidacji**: Biblioteka obsługuje zarówno proste jak i bardziej zaawansowane scenariusze walidacji, między innymi walidacja na podstawie warunków logicznych czy walidacja zagnieżdżonych obiektów.

4.4.2 Znane reguły walidacji

- `NotEmpty`(): Sprawdza czy wartość pola nie jest pusta,
- `Length`(): Określa minimalną i maksymalną długość ciągu znaków,
- `InclusiveBetween`(): Określa czy wartość znajduje się w danym zakresie,
- `Matches`(): Sprawdza czy wartość pasuje do określonego wzorca wyrażenia regularnego,
- `Equal`(): Sprawdza czy dana wartość jest równa innej podanej wartości,
- `GreaterThan`() / `LessThan`(): Sprawdza czy wartość jest większa lub mniejsza od określonej wartości,
- `Custom`(): Pozwala na zdefiniowanie niestandardowej logiki walidacji za pomocą funkcji niestandardowej.

Listing 6 ilustruje przykładowe użycie Fluent Validation w kontrolerze ASP.NET.

Listing 6 Przykład walidacji kontrolera ASP .NET z użyciem Fluent Validationn. Źródło: [26]

```
    using FluentValidation;
    using Microsoft.AspNetCore.Builder;
    using Microsoft.AspNetCore.Hosting;
    using Microsoft.Extensions.DependencyInjection;
    using Microsoft.Extensions.Hosting;

    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    public class PersonValidator : AbstractValidator<Person>
    {
        public PersonValidator()
        {
            RuleFor(person => person.Name)
                .NotEmpty().WithMessage("Name is required.")
                .Length(2, 50).WithMessage("Name must be between 2 and 50 characters.");

            RuleFor(person => person.Age)
                .InclusiveBetween(0, 120).WithMessage("Age must be between 0 and 120.");
        }
    }

    var builder = WebApplication.CreateBuilder(args);
    builder.Services.AddControllers();
    builder.Services.AddFluentValidation(fv=>fv.RegisterValidatorsFromAssemblyContaining<Pe
rsonValidator>());

    var app = builder.Build();

    app.MapControllers();

    app.Run();

    [ApiController]
    [Route("api/[controller]")]
    public class PersonController : ControllerBase
    {
        [HttpPost]
        public ActionResult<string> Create([FromBody] Person person)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            return "Person created successfully!";
        }
    }
```

4.5 JavaScript

JavaScript to dynamiczny język programowania stosowany głównie do tworzenia interaktywnych aplikacji internetowych. Jest on szeroko używany zarówno po stronie klienta (frontend), jak i serwera (backend), dzięki możliwości wykorzystania platformy Node.js.

4.5.1 Historia

Według [27] historia JavaScript sięga lat 90. XX wieku i jest ściśle związana z rozwojem przeglądarek internetowych oraz potrzebą dynamicznego tworzenia interaktywnych stron internetowych. Dzięki ciągłemu rozwojowi, JavaScript stał się jednym z najpopularniejszych języków programowania na świecie. Jego wszechstronność, łatwość nauki i szerokie wsparcie społeczności sprawiają, że jest niezastąpionym narzędziem w tworzeniu interaktywnych aplikacji internetowych oraz aplikacji serwerowych.

4.5.2 Składnia

- **Zmienne i Typy**: W JavaScript zmienne nie muszą być zadeklarowane z określonym typem przed użyciem. Typy mogą być proste lub złożone. Listing 7 zawiera przykład definiowania różnych typów zmiennych w JavaScript oraz ich wykorzystania, ilustrując elastyczność języka w obsłudze różnych struktur danych;

Listing 7 Przykład różnych zmiennych w JS. Źródło:

```
let name = "Alice";          // String
const age = 30;              // Number
let isStudent = true;        // Boolean
let grades = [90, 85, 88];   // Array
let student = {
    firstName: "Alice",
    lastName: "Smith",
    age: 30
};
console.log(typeof name);
console.log(typeof age);
console.log(typeof isStudent);
console.log(typeof grades);
console.log(typeof student);
name = "Bob";
console.log(name);
```

- **Instrukcje sterujące**: JavaScript tak jak większość języków programowania oferuje standardowe instrukcje sterujące takie jak if, else, switch oraz pętle. Listing 8 prezentuje przykład zastosowania instrukcji warunkowych oraz pętli w JavaScript, ilustrując, jak można wykorzystać te mechanizmy do podejmowania decyzji i iteracji w kodzie;

Listing 8 Przedstawienie instrukcji warunkowych w JS. Źródło: Opracowanie własne

```
let number = 10;

if (number > 0) {
    console.log("Positive number");
} else if (number < 0) {
    console.log("Negative number");
} else {
    console.log("Zero");
}

switch (number) {
    case 1:
        console.log("One");
        break;
    case 10:
        console.log("Ten");
        break;
    default:
        console.log("Not one or ten");
}
```

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}

let count = 0;
while (count < 5) {
    console.log(count);
    count++;
}
```

- **Funkcje i metody**: Funkcje są podstawowymi jednostkami strukturalnymi w JS. Po deklaracji mogą zwracać wartość lub jej nie zwracać. Listing 9 ilustruje przykłady deklaracji i używania funkcji w JavaScript;

Listing 9 Przykład deklaracji I używania funkcji w JS. Źródło:

```
function greet(name) {
    return `Hello, ${name}!`;
}

const greeting = greet("Alice");
console.log(greeting);

const numbers = [1, 2, 3, 4, 5];

const doubled = numbers.map(function(num) {
    return num * 2;
});

console.log(doubled);

const person = {
    firstName: "John",
    lastName: "Doe",
    fullName: function() {
        return `${this.firstName} ${this.lastName}`;
    }
};
console.log(person.fullName());
```

- **Kolekcje**: JavaScript oferuje różne typy kolekcji, takie jak listy, słowniki, kolejki. Ułatwiają one przechowywanie i zarządzanie danymi. Listing 10 przedstawia przykłady użycia kolekcji takich jak tablica, zestaw (*Set*), mapa (*Map*) oraz tablica z metodą `forEach`.

Listing 10 Przykład kolekcji w JS. Źródło:

```
const array = [1, 2, 3, 4, 5];
array.push(6);
console.log(array);

const set = new Set();
set.add(1);
set.add(2);
set.add(2);
set.add(3);
console.log(set);

const map = new Map();
map.set("name", "Alice");
```

```
map.set("age", 30);
console.log(map.get("name"));
console.log(map.has("age"));

const fruits = ["apple", "banana", "orange"];
fruits.forEach(function(fruit) {
    console.log(fruit);
});
```

4.5.3 Semantyka

- **Zasięg zmiennych**: Zmienne w JavaScript mogą mieć globalny lub lokalny zasięg. Zmienne zadeklarowane wewnątrz funkcji mają zasięg lokalny podczas gdy zmienne zadeklarowane poza funkcjami mają zasięg globalny. Listing 11 ilustruje ten koncept poprzez przykłady zasięgu zmiennych w funkcjach oraz w ramach bloku kodu;

Listing 11 Przykład zasięgu zmiennych w JS. Źródło:

```
function outerFunction() {
    let outerVar = "I'm outside!";

    function innerFunction() {
        let innerVar = "I'm inside!";
        console.log(outerVar);
        console.log(innerVar);
    }

    innerFunction();
    console.log(outerVar);
    // console.log(innerVar); // This would cause an error
}

outerFunction();

if (true) {
    var globalVar = "I'm a global variable!";
    let blockVar = "I'm a block variable!";
}
console.log(globalVar);
console.log(blockVar); // This would cause an error
```

- **Funkcje anonimowe i wywołania zwrotne**: JavaScript pozwala na tworzenie funkcji anonimowych i przekazywanie ich jako argumentów do innych funkcji. Jest to często używane w asynchronicznym programowaniu i obsłudze zdarzeń;
- **Hoisting**: JavaScript podnosi deklaracje zmiennych i funkcji do góry zanim zostanie wykonany kod. Jest to istotne przy zrozumieniu zachowania zmiennych i funkcji w czasie wykonywania;
- **Asynchroniczność**: JavaScript obsługuje asynchroniczność za pomocą funkcji zwrotnych (ang. *callbacks*), obietnic (ang. *promises*), oraz funkcji `async`/`await`. Pozwala to na wykonywanie operacji bez blokowania interfejsu użytkownika i zapewnia płynność działania aplikacji. Listing 12 przedstawia przykład asynchronicznego programu, w którym różne operacje są wykonywane asynchronicznie.

Listing 12 Przykład asynchronicznego program w JS. Źródło:

```
console.log("Start");

setTimeout(() => {
    console.log("Timeout 1");
```

```
}, 2000);

setTimeout(() => {
    console.log("Timeout 2");
}, 1000);

Promise.resolve().then(() => {
    console.log("Promise resolved");
});

console.log("End");
```

4.5.4 Problem wielowątkowości

Wielowątkowość w JavaScript jest dość nietypowa w porównaniu do innych języków programowania. Wynika to z asynchronicznego charakteru języka oraz modelu jednowątkowego przeglądarki. Problem wielowątkowości w JavaScript wynika głównie z faktu, że większość operacji wejścia/wyjścia (I/O) takich jak pobieranie danych z serwera, operacje na plikach czy wywołania sieciowe są wykonywane asynchronicznie co może prowadzić do blokowania interfejsu użytkownika (UI) lub opóźnień w działaniu aplikacji.

Przykładowe problemy wynikające z wielowątkowości JavaScript:

- Zablokowany interfejs użytkownika (UI),
- Opóźnienia w odpowiedziach sieciowych,
- Łączenie zależności i obsługa błędów,
- Ryzyko blokowania operacji I/O,
- Różnice w wydajności przeglądarek,
- Praca na wątkach w Node.js,

4.6 React

Według [28] React jest javascriptową biblioteką służącą do tworzenia interfejsów użytkownika. React znacznie ułatwia tworzenie interaktywnych UI. Pozwala na zaprojektowanie prostych widoków obsługujących stan aplikacji, umożliwia sprawną aktualizację i ponowne renderowanie odpowiednich komponentów.

4.6.1 Komponenty

Komponenty pozwalają podzielić interfejs użytkownika na niezależne, umożliwiające ich ponowne użycie części i myśleć o każdej z nich osobno. Według oficjalnej dokumentacji [28] koncepcyjnie, komponenty są jak javascriptowe funkcje. Przyjmują one arbitralne wartości na wejściu (nazywane “właściwościami” (ang. *props*)) i zwracają reactowe elementy opisujące, co powinno się pojawić na ekranie. Komponenty dzieli się na funkcyjne i klasowe.

4.6.2 Stany i cykl życia komponentu

1. Montowanie;
    - `constructor()`: Pierwsza metoda wywoływana podczas inicjalizacji komponentu. Jest to miejsce do ustawienia początkowych stanów i wiązań metod,

- `static getDerivedStateFromProps()`: Ta metoda pozwala na synchronizację stanu komponentu z jego właściwościami przed renderowanie,
- `render()`: Renderuje zawartość komponentu,
- `componentDidMount()`: Wywoływana po tym jak komponent zamontowany został w drzewie DOM. Jest to miejsce do pobierania danych z serwera lub subskrypcji zdarzeń.

Listing 13 przedstawia przykład cyklu życia komponentu React, który ilustruje kluczowe metody montowania komponentu.

Listing 13 Przykład cyklu życia komponentu React. Źródło: [28]

```
import React, { Component } from 'react';

class MyComponent extends Component {
    constructor(props) {
        super(props);
        this.state = {
            data: null,
            loading: true
        };
    }

    static getDerivedStateFromProps(nextProps, prevState) {
        if (nextProps.data !== prevState.data) {
            return { data: nextProps.data };
}
        return null;
    }

    componentDidMount() {
        fetch('https://api.example.com/data')
            .then(response => response.json())
            .then(data => {
                this.setState({ data, loading: false });
            });
    }

    render() {
        const { data, loading } = this.state;

        if (loading) {
            return <div>Loading…</div>;
        }

        return (
            <div>
                <h1>Data:</h1>
                <pre>{JSON.stringify(data, null, 2)}</pre>
            </div>
        );
    }
}

export default MyComponent;
```

2. Aktualizacja;
   - `static getDerivedStateFromProps()`: Wywoływana aby zaktualizować stan komponentu na podstawie nowych właściwości,

- `shouldComponentUpdate`(): Metoda która pozwala na kontrolę czy komponent powinien zostać ponownie renderowany (zwraca true lub false),
- `render`(): Komponent jest ponownie renderowany jeśli `shouldComponentUpdate`() zwróci true,
- `getSnapshotBeforeUpdate`(): Pozwala na przechwycenie pewnych informacji zanim DOM zostanie zaktualizowany,
- `componentDidUpdate`(): Wywoływana po aktualizacji komponentu, jest to miejsce do obsługi zmian w DOM lub do interakcji z komponentami potomnymi.

Listing 14 przedstawia przykład komponentu React, który ilustruje, jak działają metody cyklu życia odpowiedzialne za aktualizację komponentu.

Listing 14 Przykład reaktywnego zachowania kodu React. Źródło: [28]

```
import React, { Component } from 'react';

class MyComponent extends Component {
    constructor(props) {
        super(props);
        this.state = {
            count: 0
        };
    }

    static getDerivedStateFromProps(nextProps, prevState) {
        if (nextProps.increment !== prevState.count) {
            return { count: nextProps.increment };
        }
        return null;
    }

    shouldComponentUpdate(nextProps, nextState) {
        return nextState.count % 2 === 0;
    }

    getSnapshotBeforeUpdate(prevProps, prevState) {
        return prevState.count;
    }

    componentDidUpdate(prevProps, prevState, snapshot) {
        console.log(`Previous count: ${snapshot}`);
    }

    render() {
        return (
            <div>
                <h1>Count: {this.state.count}</h1>
                <button onClick={() => this.setState({ count: this.state.count + 1 })}>
                    Increment
                </button>
            </div>
        );
    }
}

export default MyComponent;
```

3. Odłączanie;

   - `componentWillUnmount()`: Wywoływana przed usunięciem komponentu z drzewa DOM. Jest to miejsce do czyszczenia zasobów, anulowania subskrypcji lub odłączania od zdarzeń.

Listing 15 przedstawia przykład komponentu React, który wykorzystuje metodę `componentWillUnmount()` do czyszczenia zasobów i odłączania od DOM, zanim komponent zostanie usunięty.

Listing 15 Przykład odłączenia od DOM. Źródło: [28]

```
import React, { Component } from 'react';

class TimerComponent extends Component {
    constructor(props) {
        super(props);
        this.state = {
            seconds: 0
        };
        this.timer = null;
    }
    componentDidMount() {
        this.timer = setInterval(() => {
            this.setState(prevState => ({ seconds: prevState.seconds + 1 }));
        }, 1000);
    }

    componentWillUnmount() {
        clearInterval(this.timer);
        console.log("Timer stopped and resources cleaned up.");
    }
    render() {
        return (
            <div>
                <h1>Seconds: {this.state.seconds}</h1>
            </div>
        );
    }
}

export default TimerComponent;
```

4.6.3 Routing

Routing w React jest procesem nawigacji między różnymi widokami lub komponentami w aplikacji internetowej, w zależności od aktualnego adresu URL. Jest to kluczowa część wielu aplikacji internetowych, które składają się z wielu widoków lub stron.

4.7 TypeScript

TypeScript według [30] jest językiem programowania, który powstał jako rozwinięcie języka JavaScript, dodając do niego statyczną typizację. Został stworzony przez firmę Microsoft i opublikowany na licencji open-source. TypeScript jest kompilowany do kodu JavaScript, co oznacza, że ostatecznie uruchamia się w przeglądarce lub na serwerze tak samo jak zwykły JavaScript. TypeScript dostarcza narzędzia do ułatwienia pracy programistów, takie jak IntelliSense w środowiskach programistycznych (np. Visual Studio Code), które pomagają w zwiększeniu produktywności poprzez podpowiedzi dotyczące składni, typów itp.

4.7.1 Koncept kontroli typów

TypeScript dodaje możliwość definiowania typów zmiennych, parametrów funkcji i innych elementów kodu. Dzięki temu programista może zdefiniować typy danych w swoim kodzie i uniknąć wielu błędów podczas jego pisania, ponieważ kompilator może je wykryć już na etapie kompilacji.

4.7.2 Schemat działania

Schemat pracy przy użyciu TypeScript prezentuje się następująco:

- Pisanie kodu źródłowego: Programiści piszą kod źródłowy w TypeScript, który jest rozszerzeniem języka JavaScript.
- Kompilacja kodu: Kod źródłowy napisany w TypeScript jest kompilowany do kodu JavaScript. Jest to możliwe dzięki TypeScript Compiler (TSC), który przetwarza pliki .ts (TypeScript) na pliki .js (JavaScript)
- Analiza typów: Jedną z kluczowych cech TypeScript jest jego zdolność do statycznej analizy typów. Oznacza to, że TypeScript sprawdza, czy typy zadeklarowane w kodzie źródłowym są zgodne z oczekiwanymi typami.
- Generowanie plików JavaScript: Po pomyślnej analizie i sprawdzeniu typów, TypeScript generuje równoważny kod JavaScript. Kod ten zostanie wykonany w wybranym przez programistę środowisku.

4.7.3 Typy

W TypeScript składnia „*type*” [29] umożliwia definiowanie własnych typów danych, co pozwala programistom na bardziej przejrzyste i zwięzłe definiowanie struktur danych oraz wielokrotne ich użycie w kodzie, co poprawia spójność i minimalizuje ryzyko błędów. Dodatkowo type zapewnia elastyczność w modelowaniu bardziej złożonych struktur danych oraz umożliwia kompilatorowi dokładniejsze sprawdzanie poprawności typów, co pozwala wychwytywać błędy już na etapie kompilacji. Listing 16 przedstawia przykład definiowania typu w TypeScript.

Listing 16 Przykład definicji typu. Źródło: Opracowanie własne

```
export type RestaurantType = {
 id: number;
 groupName: string;
 restaurantId: number;
 name: string;
 restaurantType: LocalType;
 address: string;
 city: string;
 isVerified: boolean;
};
```

4.7.4 Porównanie z JavaScript

TypeScript wprowadza statyczne typowanie, które pomaga programiście w wykrywaniu błędów jeszcze przed uruchomieniem kodu. Dzięki `interface` i typom parametrów możemy dokładnie określić, jakie wartości oczekujemy. W JavaScript, natomiast, tego typu zabezpieczeń nie ma – kod jest bardziej dynamiczny i podatny na błędy wynikające z nieodpowiednich typów lub struktur danych.

Listing 17 Przykład komponentu napisanego w TypeScript. Źródło: Opracowanie własne

```
    interface threadProps {
       thread: ThreadType,
       handleThreadMaximize: Function,
       renderUserPhotos: Function,
       handleThreadClose: Function
    }


    const  InactiveThread:  React.FC<threadProps>  =  ({  thread,  handleThreadMaximize,
handleThreadClose, renderUserPhotos}) => {


       const [isHovering, setIsHovering] = useState<boolean>(false)


       return (
           <Tooltip
           title={`${thread.title}`}
           placement="left"
           arrow
           >
               <div  key={thread.threadId}  className="relative  flex  items-center  justify-
center" onMouseEnter={() => setIsHovering(true)} onMouseLeave={() => setIsHovering(false)}>
                   <button onClick={() => handleThreadMaximize(thread)}>
                       {renderUserPhotos(thread)}
                   </button>
                       {
                           isHovering &&
                           <button className="" onClick={() => handleThreadClose(thread)}>
                               <CloseIcon className="h-3 w-3 p-0"/>
                           </button>
                       }
               </div>
           </Tooltip>
       )
    }


    export default InactiveThread
```

Listing 18 Przykład komponentu napisanego w JavaScript. Źródło: Opracowanie własne

```
    const   InactiveThread   =   ({   thread,   handleThreadMaximize,   handleThreadClose,
renderUserPhotos }) => {
       const [isHovering, setIsHovering] = useState(false);


       return (
           <Tooltip
               title={`${thread.title}`}
               placement="left"
               arrow
               >
               <div key={thread.threadId}
    className="relative    flex    items-center    justify-center"    onMouseEnter={()    =>
setIsHovering(true)} onMouseLeave={() => setIsHovering(false)}>
                   <button onClick={() => handleThreadMaximize(thread)}>
                       {renderUserPhotos(thread)}
                   </button>
                   {isHovering && (
                       <button
                           className=""
                           onClick={() => handleThreadClose(thread)}>
```

```
                        <CloseIcon className="h-3 w-3 p-0"/>
                    </button>
                )}
            </div>
        </Tooltip>
    );
};


export default InactiveThread;
```

Listing 17 oraz Listing 18 ukazują różnice w definiowaniu komponentu między JavaScript a TypeScript:

1.Definiowanie Typów (TypeScript vs. Brak Typów w JavaScript);

- TypeScript: TypeScript używa interface, aby zdefiniować threadProps, co jasno określa typy danych wymaganych przez komponent. Dzięki temu typy przekazywanych danych są sprawdzane na poziomie kompilacji, co zapobiega potencjalnym błędom. Na przykład, thread jest oczekiwany jako ThreadType, a handleThreadMaximize jako funkcja;
- JavaScript: JavaScript nie posiada statycznego typowania, więc interfejs threadProps jest pominięty. To oznacza, że programista musi sam dbać o prawidłowe przekazanie danych. Nie ma natychmiastowego sprawdzenia błędów wynikających z niezgodnych typów, co zwiększa ryzyko błędów;

2.Deklaracja Komponentu jako `React.FC<threadProps>`;

- TypeScript: `React.FC<threadProps>` w deklaracji komponentu określa, że `InactiveThread` jest funkcjonalnym komponentem z oczekiwanymi typami rekwizytów. Kompilator TypeScript sprawdza, czy przekazywane wartości zgadzają się z threadProps, co pozwala na dokładniejszą analizę i zapobiega problemom związanym z brakiem wymaganych rekwizytów;
- JavaScript: W JavaScript komponent jest po prostu funkcją, więc nie ma deklaracji React.FC. Programista musi pamiętać o odpowiednich rekwizytach bez wsparcia kompilatora. JavaScript nie wymusza struktury rekwizytów, co oznacza większą elastyczność, ale i możliwość błędów, jeśli przekazywane dane są niekompletne lub nieodpowiednie;

3.Typowanie `useState`:

- TypeScript: W `useState<boolean>(false)` boolean określa, że `isHovering` jest wartością logiczną. Dzięki temu typ zwracany przez `useState` jest znany z góry i kompilator będzie zgłaszał błędy, jeśli wartość stanu byłaby używana w sposób niezgodny z typem boolean.
- JavaScript: W JavaScript `useState(false)` nie definiuje typu. Typ wynika tu z wartości początkowej, ale nie jest sprawdzany przez kompilator. JavaScript pozwala na dynamiczne zmiany typu wartości w stanie, co może prowadzić do błędów, jeśli nie będzie zachowana konsekwencja.

4.8 Material UI

Material UI według [31] to biblioteka komponentów React typu open-source, która implementuje metodykę stylistyczną Material Design stworzona i wspieraną przez Google. Obejmuje ona kompleksową kolekcję przygotowanych komponentów, które są gotowe do użycia w produkcji od razu

po instalacji. Dodatkowo, oferuje zestaw opcji dostosowawczych, które pozwalają na łatwe dostosowanie wyglądu danego projektu na bazie dostępnych komponentów.

Listing 19 przedstawia przykład komponentu `RestaurantReview` napisanego z użyciem Material UI. W tym kodzie wykorzystano kilka kluczowych komponentów Material UI, takich jak `Avatar`, `Button`, `Dialog` i `Rating`.

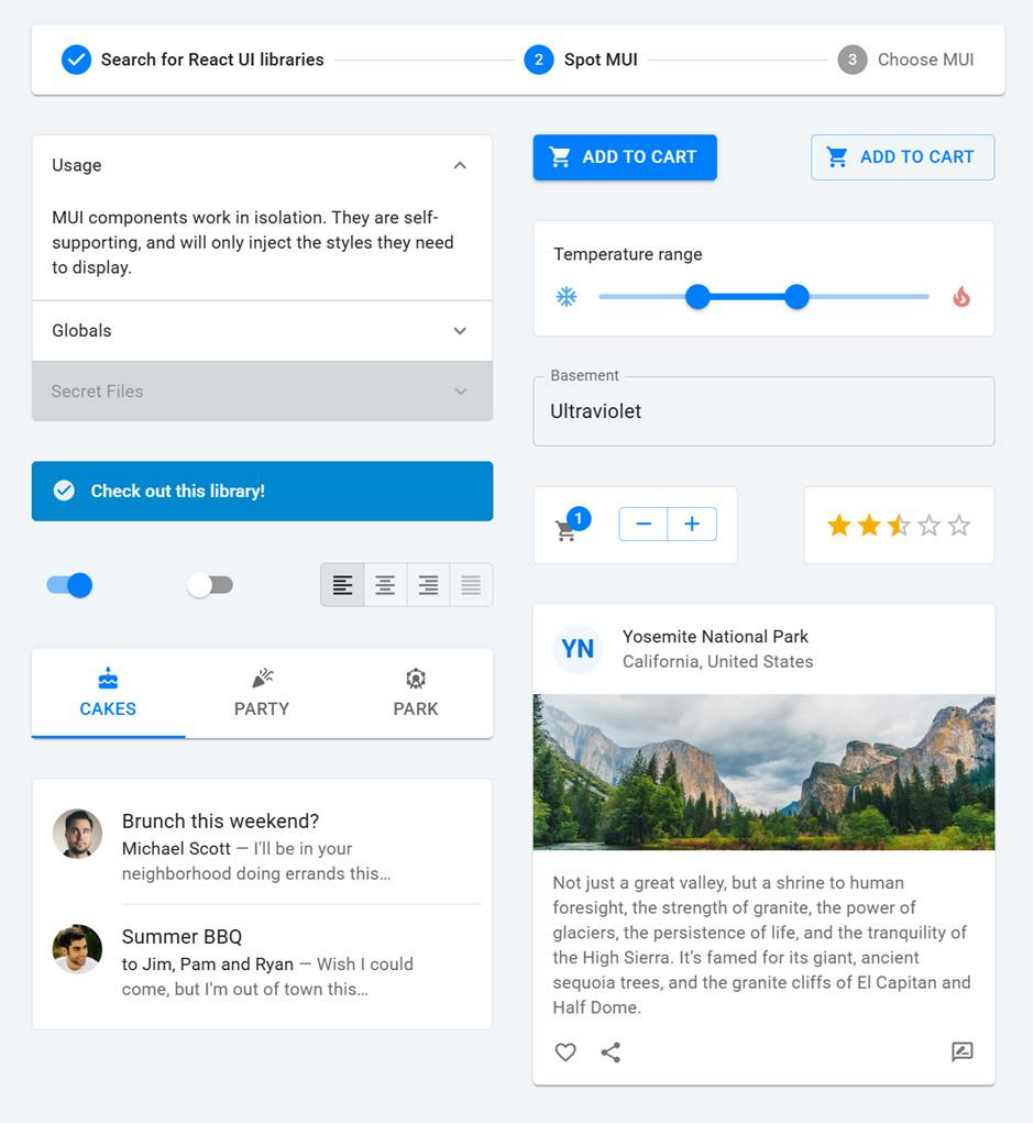
Listing 19 Przykład definicji komponentu MaterialUI. Źródło: Opracowanie własne

```
    import React, { useEffect, useState } from "react";
    import {
     Avatar,
     Button,
     Dialog,
     DialogActions,
     DialogContent,
     DialogTitle,
     Rating,
    } from "@mui/material";
    import { useTranslation } from "react-i18next";


    const RestaurantReview: React.FC = ({ /* właściwości */ }) => {
     const [isEditDialogOpen, setIsEditDialogOpen] = useState(false);
     const [isDeleteDialogOpen, setIsDeleteDialogOpen] = useState(false);
     const [isRespondDialogOpen, setIsRespondDialogOpen] = useState(false);
     const [editedStars, setEditedStars] = useState<number>(3); // Przykładowa wartość
     const [editedContents, setEditedContents] = useState<string>(""); // Przykładowa wartość
     const [responseContents, setResponseContents] = useState<string>(""); // Przykładowa
wartość
     const [t] = useTranslation("global");


     const handleEditClick = () => {
       setIsEditDialogOpen(true);
     };


     const handleDialogClose = () => {
       setIsEditDialogOpen(false);
     };


     const handleSaveEdit = async () => {
       // kod do zapisywania edytowanego przeglądu
     };


     const handleDeleteReview = async () => {
       // kod do usuwania recenzji
     };


     const handleRespondClick = () => {
       setIsRespondDialogOpen(true);
     };


     const handleSaveResponse = async () => {
       // kod do zapisywania odpowiedzi restauracji
     };

     return (
       <div className="flex flex-col gap-1 p-2 rounded-lg dark:bg-grey-6 bg-grey-0">
```

```
        <div className="flex items-center justify-between">
          <div className="flex items-center gap-4 ">
            {/* Komponent Avatar z Material-UI */}
            <Avatar alt="Author Avatar">
              {/* Przykładowe inicjały */}
              AA
            </Avatar>
            <p>Data przeglądu</p>
          </div>
          {/* Komponent Rating z Material-UI */}
          <CustomRating rating={3} readOnly={true} />
        </div>


        <div className="review-content flex flex-col items-start">
          <p>
            {/* Tekst recenzji */}
            Przykładowa treść recenzji
          </p>
        </div>


        <div className="flex items-center">
          <div className="review-actions flex items-center gap-1 ml-auto">
            {/* Komponent Button z Material-UI */}
            <Button
              id="RestaurantReviewEditButton"
              className="rounded-lg text-primary dark:text-secondary"
              onClick={handleEditClick}
            >
              Edytuj
            </Button>
            <Button
              id="RestaurantReviewDeleteButton"
              className="rounded-lg text-primary dark:text-secondary"
              onClick={() => setIsDeleteDialogOpen(true)}
            >
              Usuń
            </Button>
          </div>
        </div>

        {/* Dialog do edytowania recenzji */}
        <Dialog open={isEditDialogOpen} onClose={handleDialogClose}>
          <DialogTitle>Edytuj recenzję</DialogTitle>
          <DialogContent>
            <h2>Ocena:</h2>
            {/* Komponent Rating z Material-UI */}
            <Rating
              name="star-rating-edit"
              value={editedStars}
              onChange={(event, newValue) => setEditedStars(newValue || 3)}
            />
            <textarea
              rows={4}
              value={editedContents}
              onChange={(e) => setEditedContents(e.target.value)}
              className="w-full p-4 border rounded dark:bg-grey-700 dark:text-white
dark:border-grey-500"
            />
          </DialogContent>
          <DialogActions>
            <Button onClick={handleDialogClose} className="text-primary dark:text-
secondary">
              Anuluj
            </Button>
```

```
            <Button onClick={handleSaveEdit} className="text-primary dark:text-secondary">
              Zapisz
            </Button>
          </DialogActions>
        </Dialog>
        {/* Dialog do dodania odpowiedzi restauracji */}
        <Dialog open={isRespondDialogOpen} onClose={() => setIsRespondDialogOpen(false)}>
          <DialogTitle>Dodaj odpowiedź</DialogTitle>
          <DialogContent>
            <textarea
              rows={4}
              value={responseContents}
              onChange={(e) => setResponseContents(e.target.value)}
              className="w-[400px] p-4 border rounded dark:bg-grey-700 dark:text-white
dark:border-grey-500"
              placeholder="Wpisz odpowiedź"
            />
          </DialogContent>
          <DialogActions>
            <Button onClick={() => setIsRespondDialogOpen(false)} className="text-primary
dark:text-secondary">
              Anuluj
            </Button>
            <Button onClick={handleSaveResponse} className="text-primary dark:text-
secondary">
              Zapisz
            </Button>
          </DialogActions>
        </Dialog>
      </div>
    );
  };
  export default RestaurantReview;
```

Rysunek 39 ukazuje przykładowe komponenty dostarczone przez bibliotekę Material UI jakie mogą zostać wykorzystane przy definicji interfejsu użytkownika.



Rysunek 39 Przykładowe komponenty MaterialUI. Źródło: [32]

4.9 Jest

JEST to framework przeznaczony do testów kodu napisany w języku JavaScript. Zaprojektowany w celu zapewnienia poprawności każdej bazy kodu JavaScript. Dzięki podejściu przyjaznemu, znanemu i bogatemu w funkcje API, umożliwia pisanie testów oraz szybkie uzyskiwanie wyników. JEST jest dobrze udokumentowany, wymaga niewielkiej konfiguracji i może być rozszerzony, aby dopasować się do Twoich wymagań. Może on dodatkowo zbierać informacje o pokryciu kodu przez testy w obrębie całego projektu. Obecnie JEST jest jednym z najczęściej używanych narzędzi do testowania aplikacji Java Scriptowych na całym świecie i jest wspierany przez dużą społeczność deweloperów. Przykład testów Jest ukazuje Rysunek 40.

Rysunek 40 Przykład wykonanych testów Jest. Źródło: [34]

4.9.1 Historia

Według [33] Jest powstał jako narzędzie opracowane przez Facebooka, z myślą o testowaniu ich własnych aplikacji Java Scriptowych, zwłaszcza Reacta. Początki tego frameworka sięgają 2013 roku, kiedy to Facebook szukał efektywnego rozwiązania do testowania rosnącej bazy kodu JavaScript. Wówczas testowanie jednostkowe i integracyjne w JavaScript było trudne i czasochłonne, a istniejące narzędzia, takie jak Jasmine czy Mocha, wymagały sporej konfiguracji.

4.9.2 Założenia

Główne założenia przy projektowaniu JEST były następujące;

- Prostota – Facebook chciał stworzyć narzędzie, które mogłoby być używane bez skomplikowanej konfiguracji,
- Szybkość działania – Zwiększenie wydajności testowania, aby zminimalizować czas potrzebny na uruchamianie testów, zwłaszcza w dużych projektach,
- Kompletne narzędzie – W przeciwieństwie do innych rozwiązań dostępnych na rynku, JEST miał być "all-in-one", co oznaczało, że narzędzie nie wymagało zewnętrznych bibliotek do mockowania, asercji czy generowania raportów.

Listing 20 ilustruje przykład prostego testu komponentu React przy użyciu JEST i @testing-library/react.

Listing 20 Przykład testu komponentu React. Źródło: [35]

```
// MyComponent.js
import React from 'react';
```

```
const MyComponent = ({ name }) => {
    return <h1>Hello, {name}!</h1>;
};

export default MyComponent;

// MyComponent.test.js
import React from 'react';
import { render, screen } from '@testing-library/react';
import MyComponent from './MyComponent';

test('renders greeting message', () => {
    render(<MyComponent name="Alice" />);
    const greetingElement = screen.getByText(/Hello, Alice!/i);
    expect(greetingElement).toBeInTheDocument();
});
```

4.10 Formik

Formik według [36] to biblioteka open-source stworzona dla Reacta, która upraszcza tworzenie, walidację i obsługę formularzy w aplikacjach webowych. Głównym celem Formika jest zarządzanie stanem formularzy oraz ich walidacją w sposób prostszy i bardziej efektywny niż przy użyciu natywnego zarządzania stanem w React.

Formik oferuje szereg funkcji, takich jak:

- Zarządzanie stanem formularzy: Ułatwia pracę z polami formularza, obsługą zdarzeń (np. `onChange`, `onBlur`) oraz śledzeniem błędów walidacji i statusu wysyłki formularza,
- Walidacja: Pozwala na łatwe dodanie walidacji pól formularza za pomocą wbudowanych metod lub integrację z zewnętrznymi bibliotekami walidacyjnymi, takimi jak Yup,
- Obsługa wysyłania danych: Ułatwia wysyłanie danych z formularza, pozwalając na kontrolę tego procesu i dodanie dodatkowej logiki, np. podczas wysyłania zapytania do serwera.

Formik zapewnia deklaratywne API, które pozwala na tworzenie intuicyjnych i dobrze zarządzanych formularzy w React, redukując ilość kodu potrzebnego do obsługi typowych operacji formularzy. Listing 21 przedstawia przykład implementacji prostego formularza z wykorzystaniem Formika.

Listing 21 Przykładowy kod formularza formik. Źródło: [37]

```
<Formik
       initialValues={{ name: "", email: "" }}
       onSubmit={async (values) => {
         await new Promise((resolve) => setTimeout(resolve, 500));
         alert(JSON.stringify(values, null, 2));
       }}
     >
       <Form>
         <Field name="name" type="text" />
         <Field name="email" type="email" />
         <button type="submit">Submit</button>
       </Form>
</Formik>
```

4.11 Yup

Zgodnie z [40] Yup to biblioteka JavaScript służąca do walidacji i definiowania schematów danych. Jest szczególnie popularna w ekosystemie React, ale może być używana w dowolnej aplikacji Java Scriptowej czy Type Scriptowej. Yup pozwala na łatwe tworzenie złożonych walidacji formularzy i innych struktur danych w sposób deklaratywny i elastyczny.

4.11.1 Historia i założenia

Yup została zaprojektowana jako lekka i łatwa w użyciu alternatywa do innych narzędzi walidacji, takich jak Joi, które były bardziej popularne w środowiskach Node.js. Biblioteka została stworzona z myślą o prostocie, a jej głównym celem jest dostarczenie programistom prostego API do definiowania walidacji schematów danych w formie deklaratywnej, jednocześnie integrując się bezproblemowo z frameworkami takimi jak React i Formik.

4.11.2 Kluczowe funkcje

- **Deklaratywna walidacja schematów danych**: Yup pozwala tworzyć schematy walidacyjne dla różnych typów danych (np. stringi, liczby, obiekty, tablice). Walidacja jest deklaratywna, co oznacza, że walidacja opisywana jest w formie obiektów;
- **Łańcuchowanie walidacji**: Yup pozwala na łańcuchowanie warunków walidacyjnych, co umożliwia tworzenie bardzo złożonych reguł w prosty sposób;
- **Asynchroniczna walidacja**: Yup obsługuje walidacje asynchroniczne, co jest przydatne np. przy walidacji unikalności (np. sprawdzanie czy email jest już zajęty);
- **Walidacja złożonych struktur danych**: Yup potrafi walidować obiekty, tablice, a nawet zagnieżdżone struktury danych. Dzięki temu można łatwo walidować złożone formularze z wieloma polami;
- **Obsługa wiadomości o błędach**: Yup umożliwia personalizowanie wiadomości o błędach, co pozwala na lepsze informowanie użytkownika o błędach walidacyjnych;

Integracja z bibliotekami do zarządzania formularzami: Yup jest często używany z bibliotekami takimi jak Formik, co ułatwia zarządzanie stanem formularza i walidacją danych w aplikacjach React. Listing 22 przedstawia przykład użycia Yup do walidacji formularza.

Listing 22 Przykład użycia Yup. Źródło: [39]

```
import * as Yup from 'yup';

// Define a validation schema
const validationSchema = Yup.object().shape({
    name: Yup.string()
        .required('Name is required')
        .min(2, 'Name must be at least 2 characters long'),

    email: Yup.string()
        .email('Invalid email format')
        .required('Email is required'),

    password: Yup.string()
        .required('Password is required')
        .min(6, 'Password must be at least 6 characters long')
        .matches(/[a-zA-Z]/, 'Password can only contain Latin letters.'),

    age: Yup.number()
```

```
        .required('Age is required')
        .positive('Age must be a positive number')
        .integer('Age must be an integer')
        .min(18, 'You must be at least 18 years old'),
});

// Example function to validate data
const validateData = async (data) => {
    try {
        await validationSchema.validate(data);
        console.log('Validation successful:', data);
    } catch (error) {
        console.error('Validation error:', error.errors);
    }
};

// Example data to validate
const userData = {
    name: 'John',
    email: 'john@example.com',
    password: 'password123',
    age: 25,
};

// Validate example data
validateData(userData);
```

4.11.3 Zalety używania Yup

- Oferuje deklaratywny sposób definiowania schematów walidacyjnych, dzięki czemu kod staje się bardziej czytelny i zrozumiały;
- Tworzenie reguł walidacji w sposób elegancki i zwarty;
- Doskonała integracja z popularnymi bibliotekami do zarządzania formularzami np. Formik (płynne połączenie logiki i formularzy, łatwa obsługa błędów);
- Według [38] Yup jest w stanie obsługiwać różnorodne przypadki użycia. Oferuje rozbudowany zestaw metod do definiowania różnych reguł walidacyjnych, w tym walidacji warunkowej, asynchronicznej oraz złożonych struktur zagnieżdżonych obiektów;
- Elastyczność w zakresie dostosowywania komunikatów o błędach;
- Yup ma szczegółowo udokumentowane API, z zawarciem licznych przykładów użycia oraz aktywną społeczność, co ułatwia szybkie znajdowanie rozwiązań i wsparcia.

4.12 Kotlin

Według [41] Kotlin to język programowania, charakteryzujący się statycznym typowaniem i działaniem na maszynie wirtualnej Javy. Kotlin został stworzony jako profesjonalny język programowania obiektowego, który jest w pełni zgodny z kodem napisanym w Javie, co umożliwia firmom płynną migrację istniejącej bazy kodu.

Ten proces migracji jest ułatwiony poprzez wprowadzenie funkcji takich jak eliminacja błędów odwołania, możliwość definiowania funkcji rozszerzeń oraz wykorzystanie notacji infiksowej. Dzięki temu, przedsiębiorstwa mogą korzystać z zalet Kotlina, jednocześnie zachowując kompatybilność z istniejącym kodem i wyeliminowaniem potencjalnych problemów związanych z migracją.

Na potrzeby wsparcia dla różnych platform, twórcy języka Kotlin wydali wtyczkę do środowiska IntelliJ IDEA, która dodaje pełne wsparcie dla Kotlina. Z początku była to osobna wtyczka, jednak już od wersji 15 jest ona wbudowana w IDE jako standardowa funkcja, ułatwiając pracę programistom. Ponadto, dla użytkowników środowiska Eclipse dostępna jest dedykowana wtyczka umożliwiająca korzystanie z Kotlina. Kotlin jest kompatybilny z popularnymi narzędziami takimi jak Apache Maven, Apache Ant czy Gradle, co pozwala na łatwe integrowanie Kotlina z istniejącymi projektami i środowiskami programistycznymi. Google ogłosiło na konferencji w 2017 roku, Kotlina jako oficjalny język programowania na platformę android.

4.12.1 Historia

Zgodnie z [42] w lipcu 2011 roku firma JetBrains zaprezentowała Kotlin, czyli nowatorski język programowania dla maszyny wirtualnej Java (JVM). Dmitry Jemerov, główny programista JetBrains, podkreślił, że istniejące języki często nie spełniają oczekiwań, z wyjątkiem Scali, która jednak nie sprostała oczekiwaniom co do szybkości kompilacji. Jednym z głównych założeń projektu Kotlin było osiągnięcie czasu kompilacji porównywalnego do tego w Javie. W lutym 2012 roku JetBrains zdecydował się na otwarcie kodu projektu, udostępniając go na licencji Apache 2.0, co stanowiło istotny krok w rozwoju i promocji języka Kotlin. Pochodzenie nazwy języka Kotlin można odnaleźć w wyspie Kotlin, położonej niedaleko Petersburga

4.12.2 Kotlin vs Java

Kotlin powstał z myślą o zastąpieniu Javy i rozwiązuje szereg jej problemów. Dzięki systemowi typów w Kotlinie odwołania do null są kontrolowane, zapobiegając błędom. Kotlin eliminuje również typy generyczne, a tablice nie pozwalają na przypisanie różnych typów. W porównaniu do konwersji SAM w Javie, Kotlin oferuje odpowiednie typy funkcji. Kotlin pozwala też na wariancję w miejscu użycia bez konieczności stosowania znaków wieloznacznych. Kolejną różnicą jest brak sprawdzanych wyjątków w Kotlinie. Kotlin wprowadza również oddzielne interfejsy dla kolekcji tylko do odczytu i modyfikowalnych, co poprawia bezpieczeństwo i czytelność kodu.

Niemniej Java posiada kilka funkcji, których nie ma Kotlin. Wśród nich możemy wymienić takie jak: wyjątki kontrolowane, Java ma wyjątki kontrolowane, które wymagają jawnego obsługi lub deklaracji w sygnaturze metody, podczas gdy Kotlin nie ma wyjątków kontrolowanych. Java obsługuje statyczne składowe, ale w Kotlinie statyczne składowe są zastępowane przez obiekty towarzyszące, funkcje na najwyższym poziomie, funkcje rozszerzeń lub @JvmStatic. Listing 23 oraz Listing 24 ilustrują omówione przykłady w praktyce.

Listing 23 Implementacja klasy Person w Java. Źródło: [43]

```
class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String toString() {
        return "Name: " + name + ", Age: " + age;
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person("Alice", 30);
        System.out.println(person);
    }
}
```

Listing 24 Implementacja klasy Person w Java. Źródło:[43]

```
data class Person(val name: String, val age: Int)

fun main() {
    val person = Person("Alice", 30)
    println(person)
}
```

4.12.3 Kolin vs inne języki programowania

Kotlin jest bardziej nowoczesny w porównaniu do C++ czy C#. Eliminuje takie problemy jak niebezpieczne wskaźniki czy ręczne zarządzanie pamięcią. Ułatwia programowanie funkcyjne oraz przetwarzanie kolekcji danych oferując wbudowane funkcje takie jak lambdy, rozszerzenia funkcjonalności oraz obsługę strumieni.

W porównaniu do Pythona czy JavaScript, Kotlin oferuje statyczne typowanie, co ułatwia wykrycie błędów na etapie kompilacji oraz refaktoryzację, wpływa też na lepsze zrozumienie kodu.

4.12.4 Zastosowanie

Kotlin znalazł szerokie zastosowanie zarówno w środowiskach korporacyjnych oraz różnych organizacjach. Wśród największych i najbardziej znanych firm używających tego języka wyróżniamy: Google, Amazon Web Services, Netflix, Uber, Pinterest czy Trello. Znaczącym czynnikiem wpływającym na wzrost popularności Kotlina był wybór go przez firmę Google jako oficjalny język programowania do tworzenia aplikacji mobilnych na platformę Android.

4.13 Android SDK

Według [44] Android SDK (ang. *Android Software Development Kit)* to oficjalny zestaw narzędzi udostępniany bezpłatnie przez Google, niezbędny do tworzenia aplikacji i gier na platformę mobilną Android. Zawiera on wszystkie niezbędne elementy do projektowania, programowania, testowania, kompilowania, debugowania i publikowania aplikacji na urządzenia z Androidem.

SDK charakteryzuje się modułową budową, co oznacza, że można go rozszerzać o dodatkowe komponenty i biblioteki w zależności od potrzeb danego projektu. Dostępny jest dla systemów Windows, Linux i macOS.

4.13.1 SDK Tools

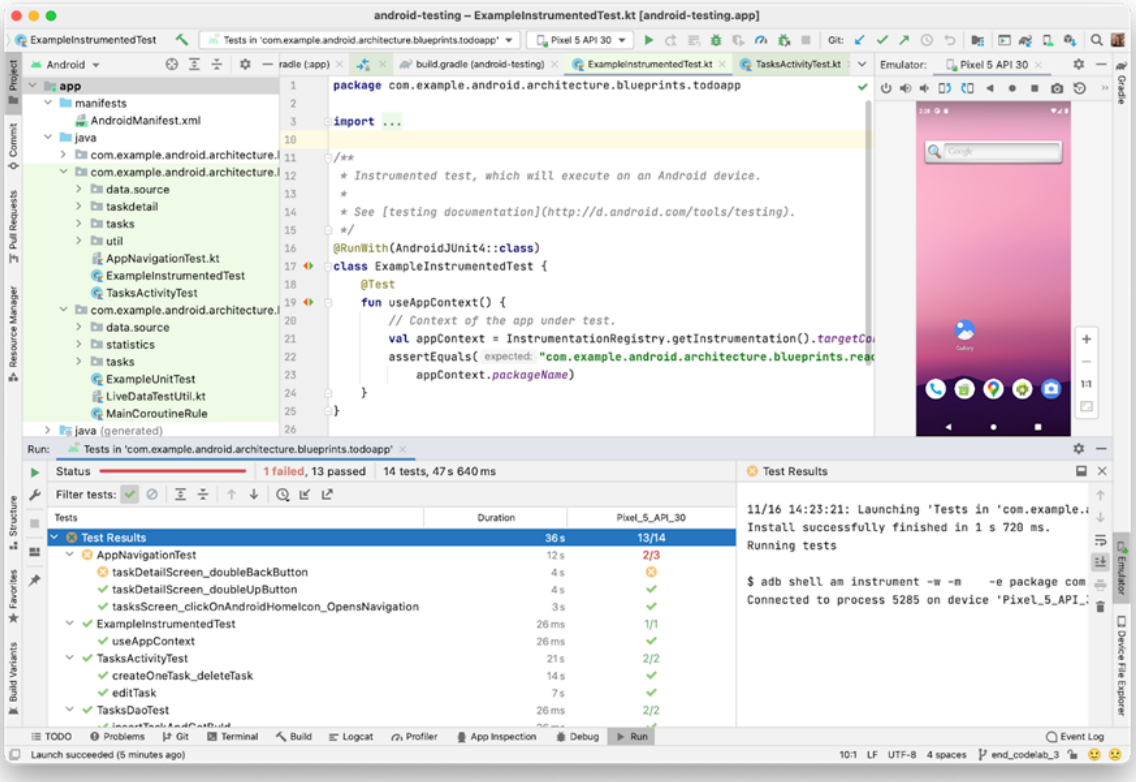
SDK Tools to podstawowa część Android SDK, która zawiera narzędzia niezbędne do tworzenia aplikacji niezależnie od wersji Androida. SDK Tools zawiera narzędzia takie jak:

- Emulator urządzenia,
- AVD Manager - zarządza wirtualnymi maszynami,

- Debuger aplikacji,
- layoutopt - analizuje rozmieszczenie komponentów na ekranie,
- sqlite3 - uzyskuje dostęp do baz danych tworzonych przez aplikacje,
- traceview - graficzna reprezentacja logów.

4.13.2 Android Studio

Android Studio to oficjalne zintegrowane środowisko programistyczne (IDE) do tworzenia aplikacji na Androida. Jest oparte na oprogramowaniu JetBrains IntelliJ IDEA. Dzięki Android Studio programiści mogą tworzyć, testować i uruchamiać aplikacje na platformie Android przy użyciu języka Kotlin oraz Java. Narzędzie korzysta z systemu kompilacji Gradle, oferuje edytor kodu, emulatory, szablony kodu, czyli pełną integrację z Android SDK. Android Studio jest oficjalnym środowiskiem programistycznym firmy Google dla systemu operacyjnego Android. Interfejs użytkownika Android studio ukazuje Rysunek 41.



Rysunek 41 Interfejs użytkownika Android Studio. Źródło: [45]

4.14 Osmdroid

Osmdroid to open-source'owa biblioteka dla systemu Android, która pozwala developerom na integrowanie danych z OpenStreetMap (OSM) do swoich aplikacji. Udostępnia widok mapy działający podobnie do Map Google, ale z tą zaletą, że jest darmowy i bazuje na kolaboracyjnych danych OSM. Osmdroid oferuje również funkcje takie jak wyświetlanie map w trybach online i offline, dodawanie nakładek na znaczniki, śledzenie lokalizacji i kształty, co czyni go wszechstronnym narzędziem do tworzenia własnych aplikacji mapowych.

4.15 Truth

Według [46] Truth to biblioteka stworzona przez Google, służąca do walidacji w testach jednostkowych i integracyjnych. Jest ceniona za swoją przejrzystość, precyzyjne komunikaty błędów oraz wygodne API. Truth umożliwia programistom pisanie testów jednostkowych i integracyjnych w sposób, który jest łatwy do zrozumienia i konserwacji. Jest alternatywą dla popularnych frameworków takich jak JUnit czy AssertJ, oferując bardziej czytelne, przejrzyste i intuicyjne API. Dzięki niej testy są bardziej zrozumiałe i łatwiejsze w utrzymaniu.

Składnia Truth opiera się na naturalnym języku, co pozwala na tworzenie testów łatwych do zrozumienia. W przykładzie, który przedstawia Listing 25 widać, jak używa się składni Truth do weryfikacji różnych warunków w testach jednostkowych.

Listing 25 Przykład użycia Truth. Źródło: [46]

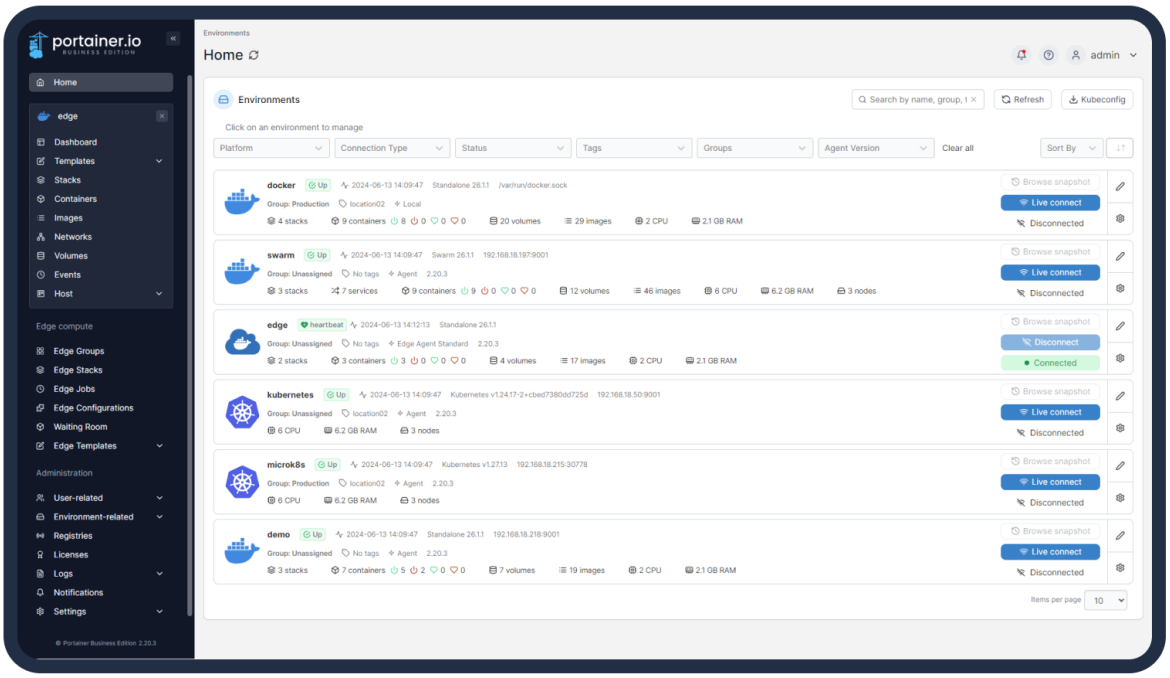
```
String string = "awesome";
assertThat(string).startsWith("awe");
assertWithMessage("Without me, it's just aweso")
    .that(string)
    .contains("me");

Iterable<Color> googleColors = googleLogo.getColors();
assertThat(googleColors)
    .containsExactly(BLUE, RED, YELLOW, BLUE, GREEN, RED)
    .inOrder();
```

4.16 Docker

Według [47] Docker to platforma informatyczna umożliwiająca automatyzację procesu wdrażania aplikacji poprzez wykorzystanie kontenerów. Kontenery w Dockerze stanowią lekkie, przenośne środowisko uruchomieniowe, które zawiera aplikację oraz wszystkie jej zależności, umożliwiając spójne działanie na różnych środowiskach. Dzięki Dockerowi, programiści mogą łatwo izolować aplikacje, zapewniając jednolite i powtarzalne środowiska deweloperskie, testowe i produkcyjne. Dodatkowo, Docker umożliwia elastyczne skalowanie aplikacji oraz efektywne zarządzanie zasobami systemowymi.

4.17 Portianer

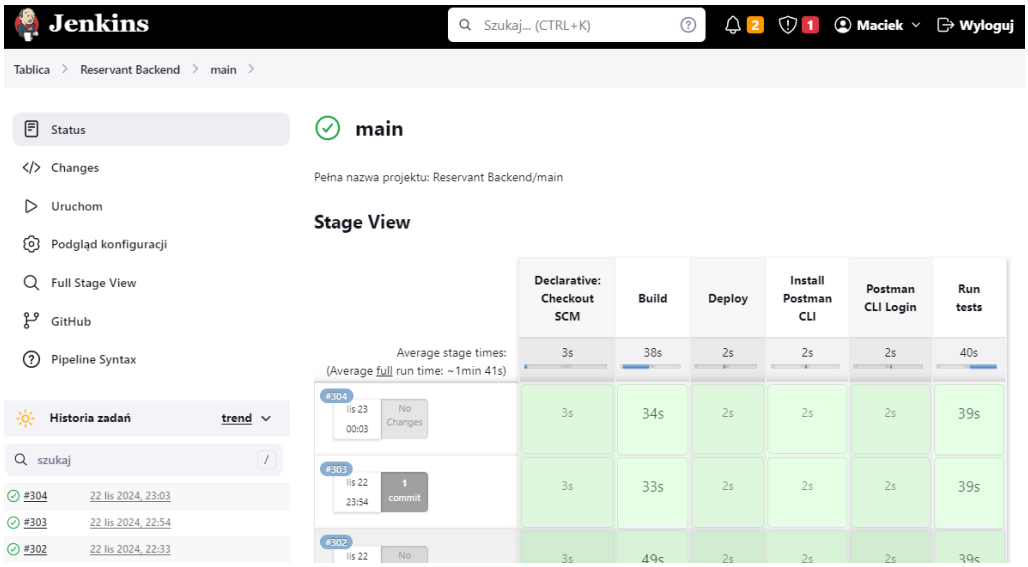
Według [48] Portainer to platforma *open-source* do zarządzania kontenerami, która dostarcza intuicyjny interfejs graficzny do monitorowania, konfiguracji i wdrażania aplikacji w środowiskach opartych na Dockerze. Jest narzędziem przeznaczonym dla administratorów systemów oraz deweloperów, umożliwiającym łatwe zarządzanie kontenerami, tworzenie sieci oraz woluminów, a także kontrolę nad zasobami systemowymi. Portainer upraszcza procesy związane z administracją kontenerami, umożliwiając skuteczne zarządzanie infrastrukturą opartą na Dockerze. Interfejs użytkownika aplikacji Portainer ukazuje Rysunek 42.



Rysunek 42 Interfejs użytkownika w aplikacji Portainer. Źródło: [48]

4.18 Jenkins

Zgodnie z [49] Serwer Jenkins to typu open source służący do automatyzacji związanej z tworzeniem oprogramowania. W szczególności ułatwia budowanie, testowanie i wdrażanie aplikacji, czyli umożliwia rozwój oprogramowania w trybie ciągłej integracji i ciągłego dostarczania. Jenkins działa na serwerach Java takich jak Apache Tomcat, Jetty itp, ale umożliwia budowanie aplikacji napisanych w wielu językach programowania. Do pobierania kodu używa standardowo różnych narzędzi kontroli wersji, w tym m.in. Subversion, Git, Mercurial, Perforce, ClearCase i AccuRev. Do budowania ma wbudowaną obsługę projektów opartych na Apache Ant, Apache Maven i sbt, ale może również wykonywać dowolne skrypty powłoki (Unix/Linux) i polecenia wsadowe systemu Windows. Panel automatyzacji w Jenkinsie przedstawia Rysunek 43.



Rysunek 43 Przykładowy automatyzacja w Jenkinsie. Opracowanie własne.

4.18.1 Integracja:

Ciągła integracja (ang. *Continous Integration*) polega na częstym umieszczaniu fragmentów kodu w repozytorium. Następnie kod ten jest budowany, a dalej uruchamiane są testy. W przypadku błędu programista szybko dostaje informację zwrotną. Najczęściej proces ten kontrolowany jest przez system ciągłej integracji. Gdy integracja odbywa się kilka razy dziennie ogranicza to ilość błędów i kosztów związanych z ich odnalezieniem.

4.18.2 Dostarczanie:

Ciągłe dostarczanie (ang. *Continous Delivery*) to automatyczna faza dostarczenia następująca po fazie integracji i przetestowaniu programu. Polega ona na przygotowaniu aplikacji do instalacji, spakowaniu i umieszczeniu w centralnym repozytorium. Dzięki takiemu podejściu zespół deweloperski może regularnie przesyłać najnowszą wersję aplikacji do innych działów w firmie. Na przykład dział marketingu może zaprezentować taki program na spotkaniu z klientem. Inne zespoły mogą wykorzystać też dostarczoną aplikację jako część większego systemu.

4.18.3 Wdrażanie:

Ciągłe wdrażanie (ang. *Continous Deployment*). Ten koncept dotyczy szczególnie (choć nie tylko) aplikacji webowych. Dzięki zastosowaniu ciągłego wdrożenia aplikacja, po fazie automatycznych testów i przygotowaniu do instalacji, może zostać automatycznie zainstalowana na serwerze produkcyjnym.

4.19 Git

Według [50] Git jest rozproszonym systemem kontroli wersji służącym do zarządzania historią kodu źródłowego, pozwalającym na niezależne zmiany w jednym kodzie, prowadzone w rożnych gałęziach.

Jego funkcjonalność ma kilka podłoży. Korzystanie z „git" jest niewątpliwie dobrym rozwiązaniem, ponieważ:

- Pozwala na jednoczesną pracę na tym samym kodzie przez kilka osób,
- Umożliwia transferowanie i łączenie zmian z różnych branchy w jednym projekcie,
- Pozwala na pracę offline we własnym repozytorium,
- Jest szybki i wydajny,
- Wspiera istniejące protokoły sieciowe, dane można wymieniać przez: HTTP(S), FTP,
- Pozwala na efektywną pracę przy dużych projektach informatycznych,
- Ogromnymi zaletami jest przechowywanie wersji oraz możliwość rozgałęziania kodu.

Podstawowe funkcje, które udostępnia narzędzie git:

- SCM – Source Code Management, kontrola wersji,
- repozytorium – miejsce przechowywania całego projektu, wszystkich wersji kodu i historii wprowadzonych zmian,
- branch – pojedyncze odgałęzienie, jedna wersja, na której pracuje dany programista,
- clone – pozwala na skopiowanie kodu z repozytorium do własnej gałęzi,

- commit – przesyła dane z Twojej gałęzi do repozytorium,
- merge – łączy zmiany wprowadzone w poszczególnych branchach.

4.20 Postman

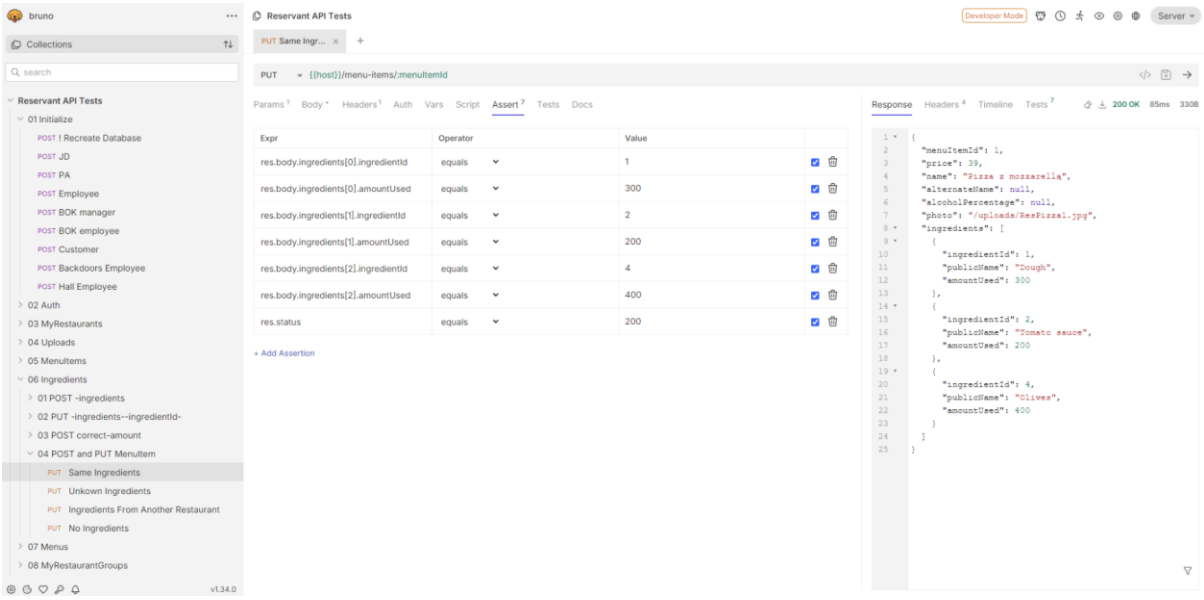
Postman to platforma zaprojektowana do testowania, rozwijania i dokumentowania API. Narzędzie to umożliwia nie tylko tworzenie zapytań HTTP, ale także analizowanie odpowiedzi co jest kluczowe dla zrozumienia jak działają różne systemy i interfejsy. Oferuje szeroki zakres funkcji, które ułatwiają pracę w zakresie testów.

Umożliwia tworzenie kolekcji, które grupują i porządkują wszystkie powiązane zapytania wraz z niezbędnymi danymi. W momencie, gdy stworzymy potrzebne kolekcje, będziemy w stanie udostępnić je wraz w wynikami naszej pracy innym użytkownikom w naszym zespole. Dodatkowo, dzięki wbudowanym narzędziom jesteśmy w stanie stworzyć automatyczne testy pozwalające na zmniejszenie liczby nieprzewidzianych błędów.

Według [51] Postman jest używany przez ponad 500 tysięcy firm na całym świecie, a dzięki swojej popularności, oferuje stałe wsparcie i rozwija swoje usługi.

4.21 Bruno

Według [52] Bruno to otwarto źródłowe narzędzie do testowania API, które jest funkcjonalną alternatywą dla Postmana. Zaprojektowane z myślą o szybkości i efektywności, pozwala na wykonywanie zapytań HTTP, testowanie punktów końcowych oraz zarządzanie kolekcjami API w sposób intuicyjny i wygodny. Bruno to świetny wybór dla deweloperów i testerów szukających wydajnego, lekkiego i elastycznego narzędzia do pracy z API. Interfejs użytkownika aplikacji Bruno przedstawia Rysunek 44.



Rysunek 44 Interfejs użytkownika w aplikacji Bruno. Źródło: Opracowanie własne.

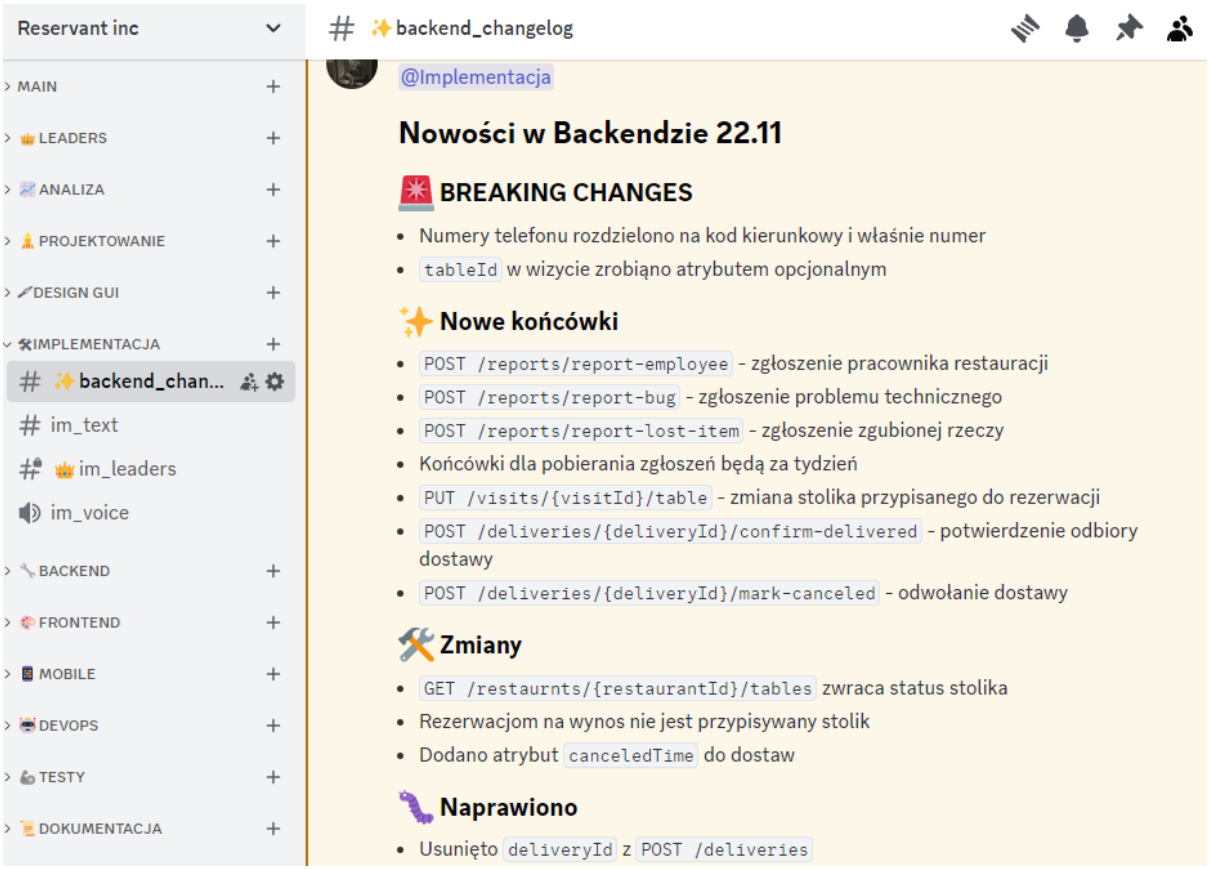
Do kluczowych cech bruno należą:

- Otwarto źródłowość – brak opłat licencyjnych i możliwość dostosowania narzędzia do własnych potrzeb,

- Obsługa kolekcji API – zarządzanie zestawami zapytań w sposób przejrzysty, z możliwością edycji i udostępniania,
- Praca offline – lokalne przechowywanie danych i brak potrzeby połączenia z internetem do działania,
- Współpraca z JSON – łatwa obsługa i modyfikacja ładunków JSON w zapytaniach i odpowiedziach,
- Zarządzanie zmiennymi – dynamiczne zmienne umożliwiają przechowywanie kluczy API, tokenów czy parametrów, co ułatwia automatyzację.

4.22 Discord

Discord zgodnie z [53] to platforma komunikacyjna, która umożliwia użytkownikom czatowanie tekstowe, głosowe i wideo w ramach serwerów tematycznych. Została stworzona głównie z myślą o graczach, ale obecnie jest popularna w wielu społecznościach, takich jak edukacja, hobby czy biznes. Użytkownicy mogą tworzyć i dołączać do serwerów, na których znajdują się różne kanały, podzielone według tematów lub form komunikacji. Discord oferuje także funkcje, takie jak boty automatyzujące działania, transmisje na żywo i integrację z innymi aplikacjami. Rysunek 45 przedstawia serwer na discordzie, który był główną platformą komunikacyjną podczas realizacji projektu.

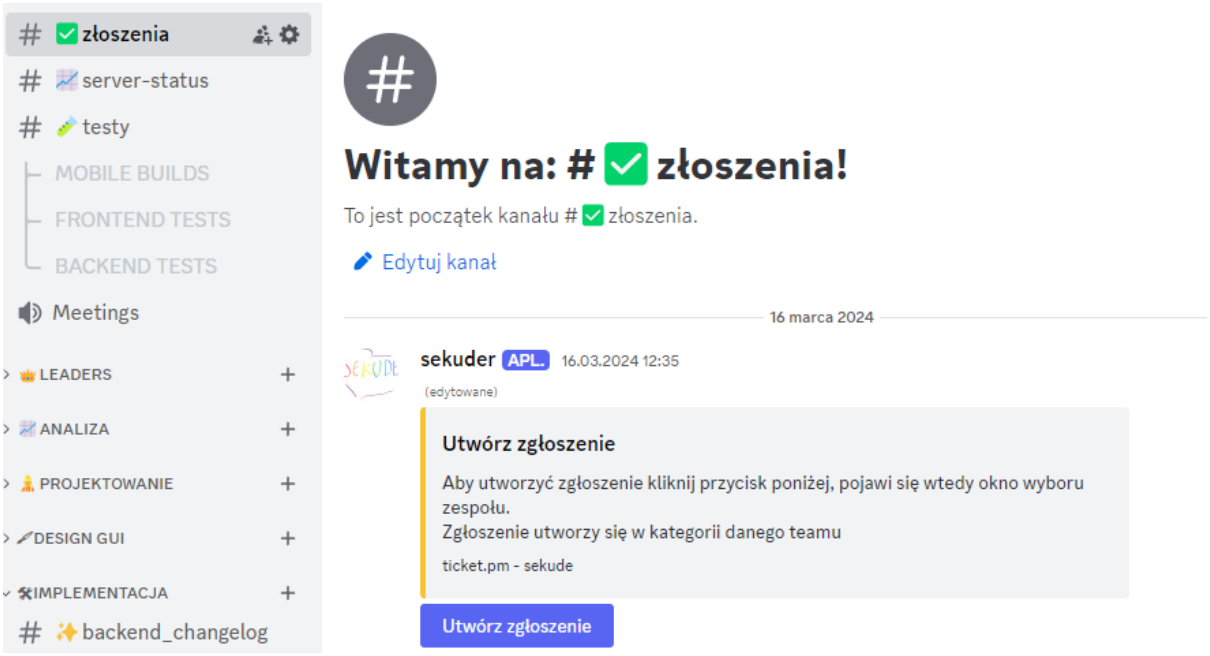


Rysunek 45 Przykładowy serwer Discord. Źródło: Opracowanie własne

W przypadku omawianego systemu stanowi on kluczowy środek komunikacji między zespołami. Do szczególnie przydatnych funkcji należą:

- Współdzielenie plików – użytkownicy mogą przesyłać dokumenty, obrazy czy linki bezpośrednio na serwerze. Dodatkowo każda wiadomość może zostać “przypięta” aby mieć do niej łatwy dostęp później;
- Wideokonferencje i rozmowy głosowe – przydatne do spotkań zespołowych i szybkich konsultacji. Możliwe jest również udostępnianie ekranu;
- Role i uprawnienia – administratorzy mogą przydzielać role, co pozwala kontrolować dostęp do poszczególnych funkcji czy kanałów;
- Integracje z narzędziami zewnętrznymi – np. Trello, GitHub czy Google Drive, co ułatwia synchronizację pracy;
- Boty – automatyzują zadania, takie jak przypomnienia, planowanie spotkań czy tworzenie zgłoszeń.

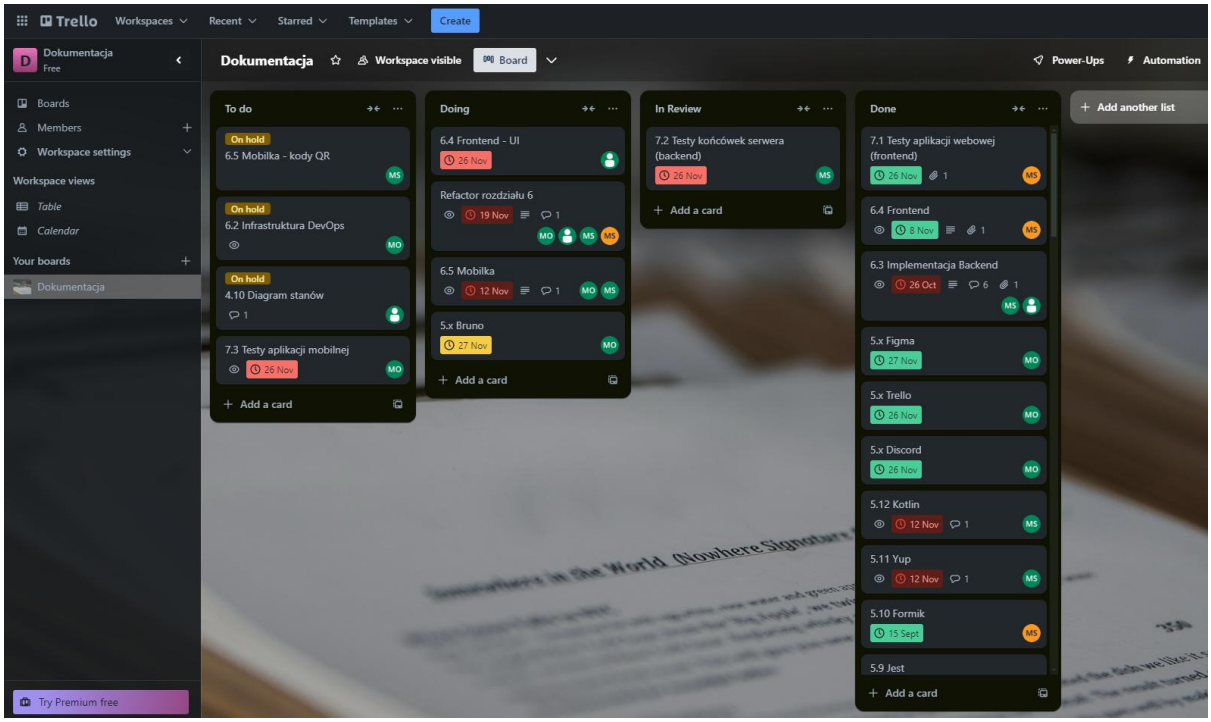
System tworzenia zgłoszeń z wykorzystaniem bota przedstawia Rysunek 46.



Rysunek 46. Przykład automatyzacji Discord. Źródło: Opracowanie własne.

4.23 Trello

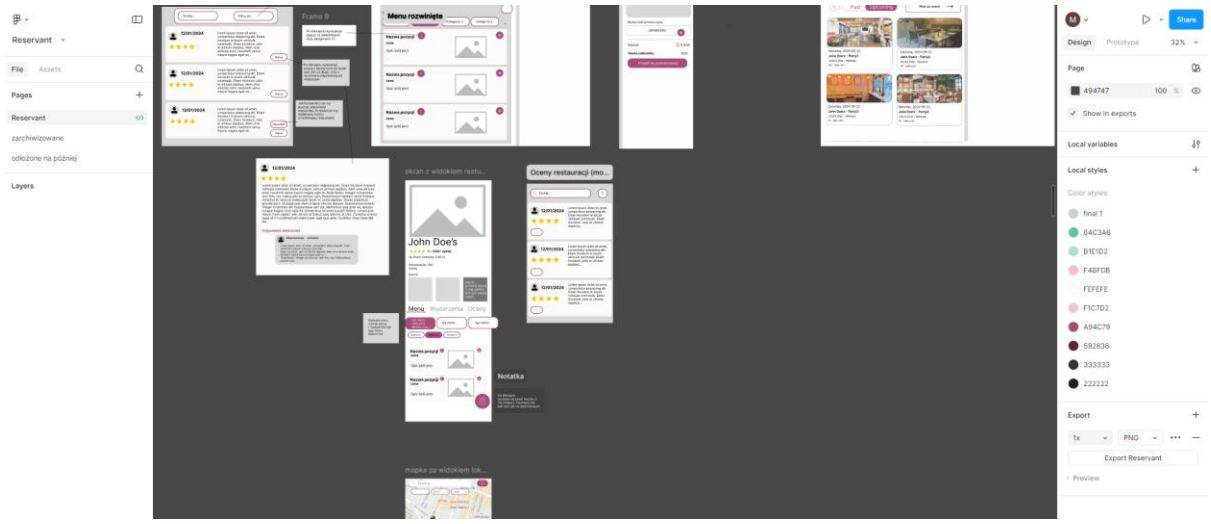
Według [54] Trello to proste i intuicyjne narzędzie do zarządzania projektami, które pozwala organizować zadania w formie tablic, list i kart. Umożliwia zespołom śledzenie postępu prac, przypisywanie zadań, ustalanie terminów i prowadzenie dyskusji w jednym miejscu. Dzięki integracjom z innymi aplikacjami oraz funkcji automatyzacji, Trello ułatwia zarządzanie nawet złożonymi projektami. Wygląd aplikacji Trello ukazuje Rysunek 47.



Rysunek 47 Przykładowe tablice na Trello. Źródło: Opracowanie własne.

4.24 Figma

Według [55] Figma to zaawansowane narzędzie do projektowania interfejsów użytkownika i współpracy zespołowej w czasie rzeczywistym, które działa w przeglądarce internetowej. Jest ceniona za swoją elastyczność i możliwość pracy w chmurze, co eliminuje potrzebę instalacji oprogramowania i umożliwia dostęp z dowolnego miejsca. Dzięki temu zespoły projektowe i programistyczne mogą pracować razem nad jednym projektem, widząc zmiany w czasie rzeczywistym. interfejs użytkownika w programie figma przedstawia Rysunek 48.



Rysunek 48 Interfejs użytkownika w programie Figma. Źródło: Opracowanie własne.

Figma to bardzo zaawansowane narzędzie. Umożliwia ona wiele przydatnych przy pracy zespołowej funkcji takich jak:

- Projektowanie interfejsów – intuicyjne narzędzia do tworzenia prototypów i makiet aplikacji lub stron internetowych,
- Współpraca w czasie rzeczywistym – możliwość jednoczesnej pracy wielu osób nad tym samym projektem, z widocznymi na żywo zmianami oraz kursorami,
- Komentarze i uwagi – użytkownicy mogą dodawać notatki bezpośrednio na projekcie, co ułatwia komunikację,
- Prototypowanie – narzędzia do tworzenia interaktywnych prototypów do testowania i prezentacji,
- Zasoby współdzielone – możliwość korzystania z bibliotek komponentów i stylów w całym zespole,
- Dostęp online – działa w chmurze, więc nie wymaga instalacji ani skomplikowanej konfiguracji.

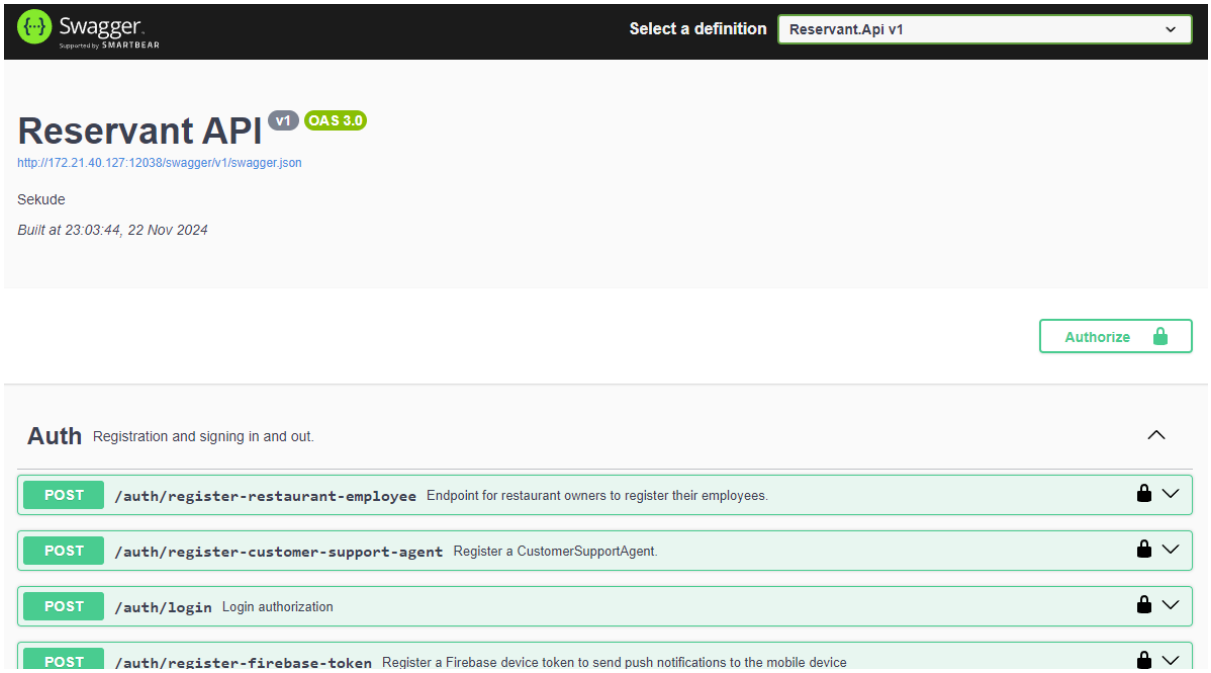
4.25 Swagger

Według [56] Swagger to zestaw narzędzi do tworzenia, dokumentowania i testowania interfejsów REST API. Głównym celem Swaggera jest ułatwienie zarządzania i komunikacji między różnymi zespołami programistycznymi przy projektowaniu API.

4.25.1 Komponenty

- Swagger Editor: Jest to narzędzie do tworzenia specyfikacji API w formacie YAML lub JSON. Umożliwia programistom definiowanie ścieżek, parametrów, operacji HTTP i innych elementów interfejsu API;
- Swagger UI: Jest to interaktywne środowisko, które generuje dokumentację interfejsu API na podstawie specyfikacji stworzonej w Swagger Editor. Użytkownicy mogą przeglądać dokumentację, testować zapytania API bezpośrednio w przeglądarce i uzyskiwać odpowiedzi w czasie rzeczywistym;
- Swagger Codegen: Narzędzie to generuje kod klienta i serwera w wielu językach programowania na podstawie specyfikacji technologii Swagger. Pozwala to programistom szybko rozpocząć pracę z nowym interfejsem REST API, eliminując potrzebę pisania kodu od zera;
- Swagger Inspector: Jest to narzędzie do testowania interfejsów REST API. Umożliwia programistom wysyłanie zapytań do API, sprawdzanie odpowiedzi oraz analizowanie wydajności i poprawności działania interfejsu;
- SwaggerHub: Jest to platforma do hostowania, udostępniania i zarządzania specyfikacjami Swagger. Umożliwia współpracę między zespołami, kontrolę wersji specyfikacji oraz integrację z systemami kontroli wersji takimi jak Git.

Przykład wykorzystania aplikacji Swagger do projektu Reservant przedstawia Rysunek 49.



Rysunek 49 Przykład użycia aplikacji Swagger. Źródło: Opracowanie własne.

4.26 Selenium

Według [57] Selenium to popularny zestaw narzędzi do automatyzacji testów aplikacji internetowych. Głównym celem Selenium jest umożliwienie programistom testowanie interfejsów użytkownika aplikacji webowych poprzez automatyzację operacji wykonywanych w przeglądarce internetowej.

4.26.1 Komponenty

- **Selenium WebDriver**: Jest to kluczowy komponent technologii Selenium który umożliwia sterowanie różnymi przeglądarkami internetowymi (takimi jak Chrome, Firefox, Safari) z poziomu kodu programistycznego. WebDriver umożliwia otwieranie przeglądarek, nawigację po stronach internetowych, wypełnianie formularzy, klikanie w elementy, pobieranie danych oraz wykonywanie innych funkcjonalności użytkownika;
- **Selenium IDE**: Jest to narzędzie do nagrywania i odtwarzania testów w przeglądarce. Selenium IDE pozwala programistom nagrywać swoje interakcje z aplikacją webową, następnie odtwarzać je automatycznie w celu sprawdzenia poprawności działania aplikacji;
- **Selenium Grid**: Jest to komponent technologii Selenium który umożliwia równoległe wykonywanie testów na wielu maszynach w różnych przeglądarkach jednocześnie. Selenium Grid pozwala na zwiększenie wydajności testów oraz umożliwia sprawdzenie czy aplikacja działa poprawnie w różnych przeglądarkach i platformach;
- **Selenium Client API**: Jest to zestaw bibliotek programistycznych dostępnych w różnych językach programowania (takich jak Java, Python, C#) które umożliwiają programistom pisanie testów automatycznych do API Selenium dostarcza różne metody i funkcje do manipulowania przeglądarką i interakcji z elementami strony internetowej.

Listing 26 przedstawia przykład prostego testu automatycznego przy użyciu Selenium w języku programowania Python.

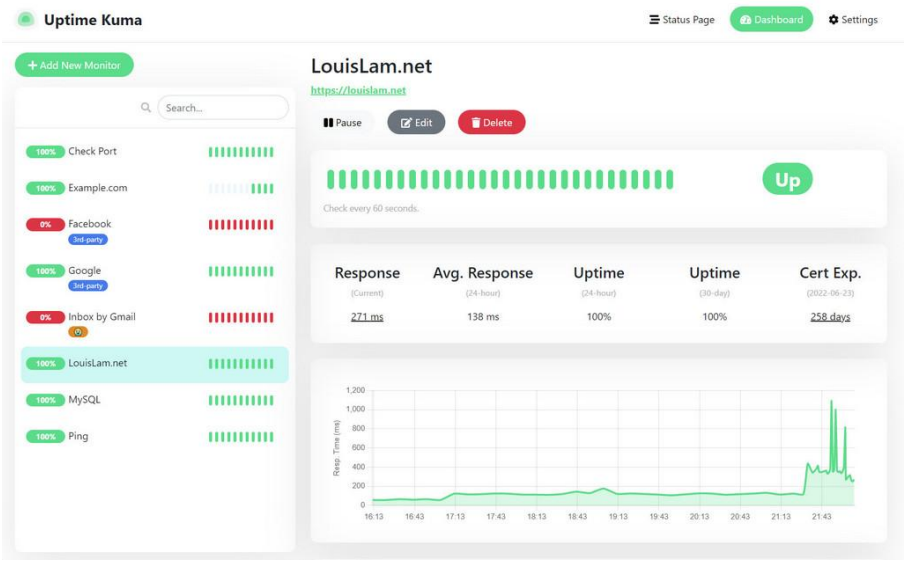
Listing 26 Przykład testu przy użyciu Selenium. Źródło: [58]

```
import unittest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.common.by import By

class PythonOrgSearch(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def test_search_in_python_org(self):
        driver = self.driver
        driver.get("http://www.python.org")
        self.assertIn("Python", driver.title)
        elem = driver.find_element(By.NAME, "q")
        elem.send_keys("pycon")
        elem.send_keys(Keys.RETURN)
        self.assertNotIn("No results found.", driver.page_source)


    def tearDown(self):
        self.driver.close()

if __name__ == "__main__":
    unittest.main()
```

4.27 Uptime Kuma

Według [59] Uptime Kuma to nowoczesne, open-source'owe narzędzie do monitorowania dostępności serwerów, usług i stron internetowych. Pozwala użytkownikom na łatwe śledzenie statusu różnych zasobów w czasie rzeczywistym oraz otrzymywanie powiadomień o problemach, takich jak awarie czy spowolnienia.

Uptime Kuma oferuje intuicyjny interfejs użytkownika, obsługę wielu rodzajów monitorowania (HTTP, TCP, ping i inne) oraz integrację z różnymi systemami powiadomień, np. e-mailem, Slackiem czy Telegramem. Dzięki elastyczności i łatwej konfiguracji stanowi doskonałe rozwiązanie zarówno dla indywidualnych użytkowników, jak i profesjonalnych zespołów IT. Interfejs użytkownika aplikacji Uptime Kuma przedstawia Rysunek 50.



Rysunek 50 Interfejs użytkownika aplikacji Uptime Kuma. Źródło: [60]

4.28 SmartFormat

Według [61] SmartFormat to biblioteka dla języka C#, która ułatwia dynamiczne formatowanie tekstu na podstawie dostarczonych danych. Jest potężnym narzędziem do generowania sformatowanego tekstu, pozwalającym na tworzenie zaawansowanych szablonów z możliwością interpolacji zmiennych, warunkowego formatowania i wsparcia dla różnych kultur językowych.

Kluczowe funkcje SmartFormat:

- Interpolacja i dynamiczne podstawianie danych: Pozwala na wstawianie wartości zmiennych do tekstowych szablonów w intuicyjny sposób,
- Warunkowe formatowanie: Obsługuje reguły „jeśli-wtedy” oraz logikę formatowania dla różnych przypadków,
- Wsparcie dla międzynarodowych formatów: Umożliwia formatowanie danych takich jak daty, liczby czy waluty z uwzględnieniem ustawień lokalnych,
- Skalowalność: Obsługuje zarówno proste przypadki, jak i bardziej złożone scenariusze dzięki rozszerzalności i możliwościom dostosowania,
- Formatowanie kolekcji: Umożliwia iterowanie i formatowanie kolekcji danych bez konieczności ręcznego implementowania pętli,
- Integracja z .NET: Ściśle współpracuje z wbudowanymi funkcjami formatowania w .NET, rozszerzając ich możliwości.

Przykłady zastosowań:

- Tworzenie szablonów e-maili z dynamicznie podstawianymi danymi (np. imię użytkownika, data),
- Generowanie raportów lub podsumowań z dynamicznie zmieniającymi się danymi,
- Formatowanie tekstów z warunkowymi elementami, np. różne wiadomości w zależności od stanu zamówienia.

Przykładem użycia tej biblioteki jest generowanie sformatowanego tekstu na podstawie szablonu, w którym przedstawiane są wartości zmiennych, jak pokazuje Listing 27.

Listing 27 Przykład użycia biblioteki SmartFormat. Źródło: Opracowanie własne

```
    using SmartFormat;

    var template = "Witaj {Name}, twoje zamówienie na {Product} zostało złożone {Date:yyyy-
MM-dd}.";
    var data = new { Name = "Jan", Product = "książkę", Date = DateTime.Now };

    string result = Smart.Format(template, data);
    // Witaj Jan, twoje zamówienie na książkę zostało złożone 2025-01-11.
```

5 Implementacja systemu

W tym rozdziale zostanie szczegółowo omówiony proces implementacji omawianego systemu zarządzania lokalem gastronomicznym oraz aplikacji klienckiej, umożliwiającej rezerwację wizyt w tych lokalach, z dodatkowymi funkcjami portalu społecznościowego – Reservant.

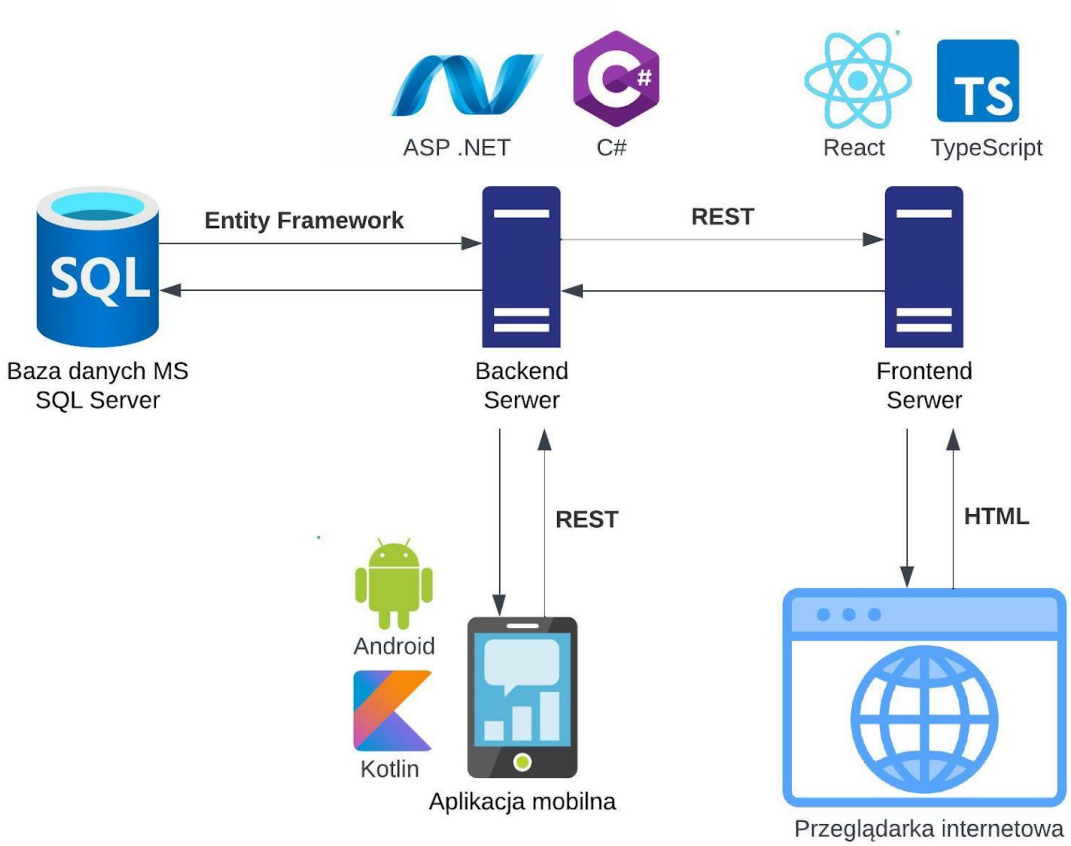
5.1 Wstęp

Nad implementacją systemu pracowały trzy główne zespoły: backend, frontend i zespół odpowiedzialny za aplikację mobilną. Każdy z nich miał jasno określone zadania, które były kluczowe dla powodzenia całego projektu.

- **Backend** – zespół ten koncentrował się na opracowywaniu logiki serwera oraz integracji wszystkich istotnych funkcjonalności po stronie serwerowej. Odpowiadał za zarządzanie danymi, komunikację z bazą danych i zapewnienie płynnego działania serwera.
- **Frontend** – ten zespół był odpowiedzialny za tworzenie części klienckiej aplikacji webowej, skupiając się na interfejsie użytkownika oraz interakcji z użytkownikiem. Ich zadaniem było zaprojektowanie i zaimplementowanie intuicyjnego oraz estetycznego interfejsu, który umożliwiłby łatwe korzystanie z aplikacji przez użytkowników.
- **Mobilka** – trzeci zespół, odpowiedzialny za stworzenie aplikacji mobilnej dostępnej na system Android. Ich praca również skupiała się na warstwie klienckiej, jednak dostosowanej do specyfiki platformy mobilnej. Zespół ten musiał uwzględniać różne wyzwania związane z mobilnością, takie jak responsywność, optymalizacja wydajności czy dostosowanie interfejsu do mniejszych ekranów.

Rysunek 51 przedstawia architekturę systemu będącego tematem tej pracy. Głównym elementem systemu jest serwer napisany w języku *C#* z użyciem technologii *ASP.NET Core*. Serwer korzysta z *Entity Framework Core* do połączenia z bazą danych *MS SQL Server*. Serwer backendowy współpracuje zarówno z serwerem aplikacji webowej, jak i aplikacją mobilną.

Komunikacja między komponentami odbywa się poprzez zapytania REST. Aplikacja mobilna, zaprojektowana dla urządzeń z systemem Android, została stworzona w języku *Kotlin*. Aplikacja webowa, napisana w *TypeScript* przy użyciu *React*, wymienia dane z przeglądarką użytkownika za pomocą zapytań HTML.



Rysunek 51 Architektura systemu Reservant. Źródło: Opracowanie własne

Większość funkcjonalności była rozwijana równocześnie zarówno w aplikacji mobilnej, jak i webowej, co zapewniało spójność obu wersji systemu. Jednakże pewne funkcje były dedykowane konkretnym grupom użytkowników. Na przykład moduł dostępny wyłącznie dla pracowników restauracji został zaprojektowany wyłącznie na potrzeby aplikacji mobilnej, aby umożliwić łatwą obsługę z poziomu urządzeń przenośnych. Równolegle powstawała aplikacja webowa przeznaczona dla pracowników Biura Obsługi Klienta (BOK), z funkcjami dostosowanymi do ich codziennych potrzeb.

Prace nad systemem były prowadzone równocześnie w poszczególnych zespołach, co znacznie przyspieszało realizację poszczególnych zadań. Dzięki temu każdy zespół mógł rozwijać swoje obszary bez zbędnych przestojów, co pozwoliło na szybkie wdrażanie kolejnych funkcji do systemu.

Już w trakcie implementacji każdej z funkcji, zespół testerów przystępował do szczegółowych testów, weryfikując zgodność aplikacji z założeniami projektowymi oraz identyfikując potencjalne problemy. Taki proces ciągłej weryfikacji pozwalał na szybkie wykrywanie i naprawianie błędów, co zminimalizowało ryzyko wystąpienia poważnych komplikacji na późniejszych etapach projektu. Dzięki temu można było uniknąć opóźnień i zapewnić terminowe wdrożenie systemu.

5.2 Implementacja infrastruktury DevOps

Odpowiednie ustawienie środowiska deweloperskiego w połączeniu z operacjami wdrożeniowymi jest kluczowe podczas tworzenia nowoczesnych systemów. W omawianym projekcie metodologia DevOps jest używana w celu zapewnienia płynności wdrożenia i monitorowania kodu.

5.2.1 Automatyczna weryfikacja kodu

Każdy z zespołów implementujących posiada swoje repozytorium na platformie github [63], w którym przechowywane są kody źródłowe. Platforma ta umożliwia również wykonywanie różnorakich zautomatyzowanych czynności wywoływanych przez konkretną akcję (np. prośba o recenzję kodu). Ustawienie takich czynności zazwyczaj odbywa się poprzez utworzenie odpowiedniego pliku yaml umieszczonego w katalogu „.github/workflows/" względem głównego folderu repozytorium. Nie inaczej jest w przypadku omawianego systemu, gdzie różne jego części są tworzone przez różne zespoły, a co za tym idzie, może to prowadzić do błędów natury ludzkiej.

Automatyczne kompilacje kodu zostały skonfigurowane, aby eliminować najbardziej podstawowe błędy już w momencie jego publikacji na platformie. Dodatkowo github wymusza poprawne przeprowadzenie takiej kompilacji zanim możliwe będzie połączenie nowego kodu z głównym branchem. Listing 28 pokazuje przykład takiej akcji dla repozytorium aplikacji mobilnej.

Listing 28 Przykład pliku konfiguracyjnego github actions. Źródło: Opracowanie własne

```
    name: Android CI

    on:
      pull_request:
        types: [opened, synchronize, reopened, closed]
        branches:
          - main

    jobs:
      build:
        runs-on: ubuntu-latest

        steps:
        - name: Checkout code
          uses: actions/checkout@v2

        - name: Cache Gradle dependencies
          uses: actions/cache@v2
          with:
            path: ~/.gradle/caches
            key: ${{ runner.os }}-gradle-${{ hashFiles('**/*.gradle*', '**/gradle-
wrapper.properties') }}
            restore-keys: |
              ${{ runner.os }}-gradle-

        - name: Set up JDK 21.0.3
          uses: actions/setup-java@v2
          with:
            java-version: '21.0.3+9.0.LTS'
            distribution: 'adopt'

        - name: Grant execute permission for gradlew
          run: chmod +x ./gradlew

        - name: Build with Gradle
          env:
              GOOGLE_SERVICES_KEY: ${{ secrets.GOOGLE_SERVICES_KEY }}
          run: ./gradlew build
```

```
      - name: Build apk debug project (APK)
        if: github.event.pull_request.merged == true
        run: ./gradlew assembleDebug

      - name: Get release file apk path
        if: github.event.pull_request.merged == true
        id: releaseApk
        run: echo "apkfile=$(find app/build/outputs/apk/debug/*.apk)" >> $GITHUB_OUTPUT

      - name: Upload Release Build to Artifacts
        if: github.event.pull_request.merged == true
        uses: actions/upload-artifact@v3
        with:
            name: release-artifacts
            path: ${{ steps.releaseApk.outputs.apkfile }}

      - name: Notify discord server
        if: github.event.pull_request.merged == true
        env:
          DISCORD_WEBHOOK_URL: ${{ secrets.DISCROD_WEBHOOK_URL }}
          ARTIFACT_ID: ${{github.run_id}}
        run: chmod +x ./notify.sh && ./notify.sh
```

Warto również zauważyć, że w tym przypadku oprócz samej kompilacji wykonywana zostaje również publikacja artefaktu w postaci pliku APK powstałego przy okazji. Plik ten opublikowany zostaje w odpowiednim miejscu na platformie komunikacyjnej jaką jest Discord.

5.2.2 Automatyczne wdrożenia

Po udanej kompilacji kodu na platformie github, kod oczekuje na wdrożenie za pośrednictwem systemu Jenkins. System ten w krótkich odstępach czasowych sprawdza, czy powstały nowe zmiany w danym repozytorium i w przypadku, gdy takie zaszły, wykonuje procedurę wdrożenia.

Instrukcje jakie czynności powinien wykonać Jenkins sprecyzowane są w pliku Jenkinsfile umieszczonym bezpośrednio w głównym katalogu repozytorium. Pliki te pisane są w języku groovy, który pozwala na uniwersalne wykonywanie poleceń. Listing 29 pokazuje przykładową implementację takiego pliku, który umieszczony jest w repozytorium aplikacji webowej.

Listing 29 Przykład pliku Jenkinsfile. Źródło: Opracowanie własne

```
pipeline {
    agent any

    stages {
        stage('Build') {
            when {
                branch 'main'
            }
            steps {
                sh "docker build -t reservant-front:latest ."
            }
        }

        stage('Deploy') {
            when {
                branch 'main'
            }
            steps {
                sh "docker stop skalmar && docker rm skalmar || true"
```

```
                    sh "docker run --detach --restart=on-failure -v
        skalmar_nginx_config:/etc/nginx/conf.d --name skalmar -p 800:80 reservant-front"
                }
            }
            stage ('Run tests') {
                when {
                    branch 'main'
                }
                steps {
                    build job: 'Reservant Frontend-Tests/main', parameters: [string(name:
'label_string', value: 'FRONTEND DEPLOY')]
                }
            }
        }
    }
```

W tym przypadku Jenkins wykorzystuje konteneryzację w dockerze jako główny środek wdrożenia. Docker każdorazowo buduje swój obraz na podstawie pliku Dockerfile (Listing 30), który również umieszczony jest w głównym folderze repozytorium. Następnie wykonywane jest polecenie "`docker run`", które wykorzystując zbudowany wcześniej obraz publikuje stronę internetową na danym porcie. Warto również zauważyć, że w tym momencie wykonywane są również testy aplikacji webowej, o których mowa w rozdziale 7.

Listing 30 Przykład pliku Dockerfile. Źródło: Opracowanie własne

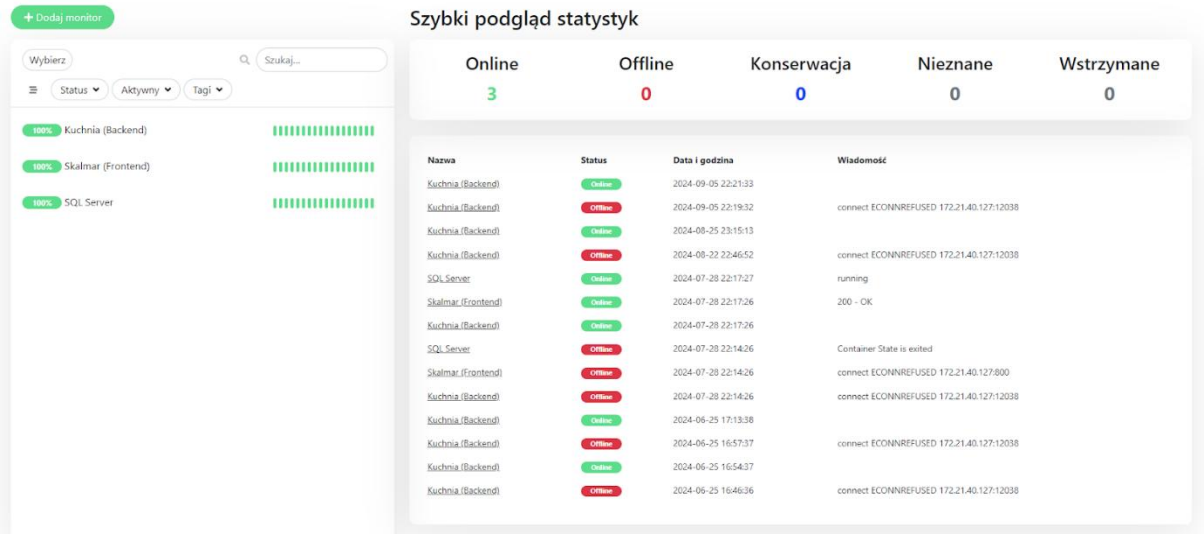
```
    # build step
    FROM node:16.13.2-alpine as build
    WORKDIR /app
    COPY package.json ./
    RUN npm install
    COPY . ./
    ENV REACT_APP_SERVER_IP $REACT_APP_SERVER_IP
    RUN npm run build

    # release step
    FROM nginx:1.21.5-alpine as release
    COPY --from=build /app/build /usr/share/nginx/html/
    EXPOSE 80
    CMD ["nginx", "-g", "daemon off;"]
```

5.2.3 Monitoring

Poprawnie skompilowane oraz wdrożone aplikacje są nieustannie monitorowane przez system zwany "Uptime Kuma". Jest to otwarto źródłowy projekt, który dostarcza bardzo podstawowe narzędzia monitorujące. Umożliwia on łatwy dostęp do historii zdarzeń dla danego środowiska.Ustawiono tam sprawdzanie serwisów kluczowych do działania przedstawionego systemu:

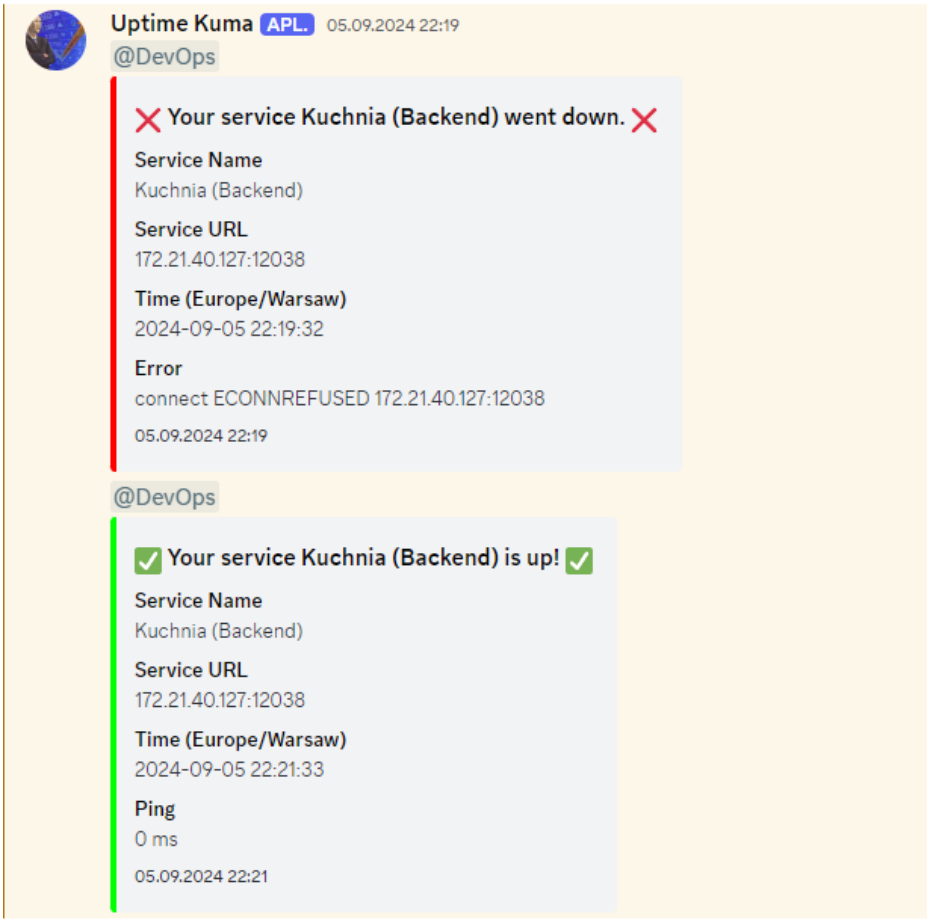
- Serwer backendowy,
- Serwer aplikacji webowej,
- Serwer bazy danych.

Wykorzystania Uptime Kuma w celu monitorowania statusu serwisów przedstawia Rysunek 52.



Rysunek 52 Panel systemu monitoringu Uptime Kuma. Źródło: Opracowanie własne

Program pozwolił ustawić również wysyłanie powiadomień na serwer Discord w przypadku, gdy którykolwiek z serwisów przestanie działać. Wysyłana jest wtedy wiadomość oznaczająca odpowiedni zespół z podaniem, który serwis przestał działać, kiedy oraz dlaczego. To działanie ukazuje Rysunek 53.



Rysunek 53 Powiadomienie wysłane przez Uptime Kuma. Źródło: Opracowanie własne

5.3 Implementacja infrastruktury serwerowej

Infrastruktura serwerowa powstała wskutek połączenia implementacji standardowej architektury ASP .NET oraz wielu bibliotek pomocniczych. Fragment ten opisuje sposób w jaki zostały zaimplementowane poszczególne części aplikacji serwerowej.

5.3.1 Architektura warstwowa

W aplikacji Reservant zastosowano architekturę warstwową, według [62], pozwalającą na przejrzyste rozdzielenie poszczególnych warstw aplikacji, co usprawniło proces implementacji, testowania oraz utrzymania kodu.

W niniejszej aplikacji widoki są dynamicznie renderowane po stronie klienta, opierając się na danych dostarczonych przez poszczególne warstwy infrastruktury serwerowej. Sposób implementacji części klienckiej oraz prezentacja przykładowych ekranów, zarówno aplikacji webowej jak i mobilnej, zostały opisane w podrozdziałach 5.4 i 5.5 oraz w rozdziale 6.

Na poszczególne warstwy struktury aplikacji serwerowej składają się:

- Warstwa komunikacji- odpowiada za komunikację z klientem, przyjmuje parametry wejściowe i wywołuje odpowiednie serwisy. Listing 31 przedstawia reprezentację obiektu menu dla klienta. Listing 32 prezentuje kod odpowiadający za przyjmowanie parametrów i wywołanie odpowiedniej metody serwisowej.

Listing 31 Przykład obiektu reprezentacyjnego. Źródło: Opracowanie własne

```
public class MenuVM
{
    public required int MenuId { get; set; }

    public required string Name { get; set; }

    public required string? AlternateName { get; set; }

    public required MenuType MenuType { get; set; }

    public required DateOnly DateFrom { get; set; }

    public required DateOnly? DateUntil { get; set; }

    public required List<MenuItemSummaryVM> MenuItems { get; set; }
}
```

Listing 32 Przykład kontrolera. Źródło: Opracowanie własne

```
/// <summary>
/// Controller for managing events
/// </summary>
[ApiController, Route("/events")]
public class EventsController(EventService service, UserManager<User> userManager) :
StrictController
{
    /// <summary>
```

```
        /// Create new event
        /// </summary>
        [HttpPost]
        [ProducesResponseType(200), ProducesResponseType(400)]
        [Authorize(Roles = Roles.Customer)]
      [MethodErrorCodes<EventService>(nameof(EventService.CreateEventAsync))]
        public async Task<ActionResult<EventVM>> CreateEvent([FromBody] CreateEventRequest
request)
        {
        var user = await userManager.GetUserAsync(User);
        if (user is null)
        {
                return Unauthorized();
        }

        return OkOrErrors(await service.CreateEventAsync(request, user));
        }
   }
```

- Warstwa logiki biznesowej- definiuje akcje, które może wykonać aplikacja. Warstwa składa się z kodu biznesowego, walidacji danych i uprawnień. Operacje wykonywane są na obiektach Warstwy danych. Listing 33 przedstawia serwis odpowiadający za zapisanie w systemie informacji o odwołaniu dostawy.

Listing 33 Przykład serwisu. Źródło: Opracowanie własne

```
    /// <summary>
    /// Service responsible for marking a delivery as canceled
    /// </summary>
    public class MarkCanceledService(
        ApiDbContext context,
        AuthorizationService authorizationService,
        IMapper mapper){
        /// <summary>
        /// Mark the delivery as canceled
        /// </summary>
        /// <param name="deliveryId">ID of the delivery</param>
        /// <param name="userId">ID of the current user</param>

[MethodErrorCodes<AuthorizationService>(nameof(AuthorizationService.VerifyRestaurantBackdoo
rAccess))]
        [ErrorCode(nameof(deliveryId), ErrorCodes.DeliveryNotPending,
            "Delivery has already been confirmed or canceled")]
        public async Task<Result<DeliveryVM>> MarkCanceled(int deliveryId, Guid userId)
        {
            var delivery = await context.Deliveries
                .Include(d => d.Ingredients)
                .ThenInclude(i => i.Ingredient)
                .SingleAsync(d => d.DeliveryId == deliveryId);
            var                      auth                      =                      await
authorizationService.VerifyRestaurantBackdoorAccess(delivery.RestaurantId, userId);
            if (auth.IsError) return auth.Errors;

            if (!delivery.IsPending)
            {
                return new ValidationFailure
                {
                    PropertyName = nameof(deliveryId),
                    ErrorCode = ErrorCodes.DeliveryNotPending,
                    ErrorMessage = "Delivery has already been confirmed or canceled",
                };
            }
```

```
            delivery.DeliveredTime = DateTime.UtcNow;
            await context.SaveChangesAsync();
            return mapper.Map<DeliveryVM>(delivery);
        }
    }
```

- Warstwa danych- niniejsza warstwa zawiera model danych, którym posługuje się aplikacja. Obiekty tej warstwy odpowiadają danym, które są przechowywane i przetwarzane. Listing 34 jest reprezentacją zamówienia w restauracji w niniejszej aplikacji.

Listing 34 Przykład obiektu z modelu danych. Źródło: Opracowanie własne

```
public class Order : ISoftDeletable
{
    public const int MaxNoteLength = 100;

    public int OrderId { get; set; }

    public int VisitId { get; set; }

    [StringLength(MaxNoteLength)]
    public string? Note { get; set; }

    public Visit Visit { get; set; } = null!;

    public ICollection<OrderItem> OrderItems { get; set; } = null!;

    public Guid? AssignedEmployeeId { get; set; } = null!;

    public User? AssignedEmployee { get; set; } = null!;

    public DateTime? PaymentTime { get; set; } = null!;

    public bool IsDeleted { get; set; }
}
```

5.3.2 JWT (JSON Web Token)

JSON Web Token jest standardem wykorzystywanym w bezpiecznym uwierzytelnianiu użytkowników w aplikacjach webowych i mobilnych. Tokeny JWT pozwalają na szybkie, bezstanowe uwierzytelnianie. Listing 35 przedstawia metodę `GenerateSecurityToken`, która generuje token zabezpieczający (`SecurityToken`) dla użytkownika. Token ten zawiera informacje o użytkowniku, takie jak ID użytkownika, nazwa użytkownika i przypisane role.

Listing 35 Kod generujący token zabezpieczając. Źródło: Opracowanie własne

```
public SecurityToken GenerateSecurityToken(User user)
{
      JwtSecurityTokenHandler handler = new();
      var claims = new List<Claim>
      {
      new(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
      new(JwtRegisteredClaimNames.Sub, user.Id.ToString()),
      new(ClaimTypes.Name, user.UserName!)
      };

      var roles = userManager.GetRolesAsync(user).Result;
      claims.AddRange(roles.Select(role => new Claim(ClaimTypes.Role, role)));
```

```
        var tokenDescriptor = new SecurityTokenDescriptor
        {
        Subject = new ClaimsIdentity(claims),
        Expires =
DateTime.UtcNow.Add(TimeSpan.FromHours(jwtOptions.Value.LifetimeHours)),
        Issuer = jwtOptions.Value.Issuer,
        Audience = jwtOptions.Value.Audience,
        SigningCredentials = new SigningCredentials(
                new SymmetricSecurityKey(jwtOptions.Value.GetKeyBytes()),
                SecurityAlgorithms.HmacSha256)
        };

        var token = handler.CreateToken(tokenDescriptor);
        return token;
    }
```

Połączenie architektury MVC z JWT stanowi mocne fundamenty zarówno w zakresie struktury aplikacji, jak i zabezpieczeń. Zapewnia to prostotę oraz bezpieczeństwo przetwarzania rezerwacji i zamówień, jak również elastyczność w zarządzaniu dostępem dla poszczególnych użytkowników systemu.

5.3.3 Baza danych i dostęp do niej

Bazę danych aplikacji Reservant utworzono przy użyciu Entity Framework Core, frameworka ORM (ang. *Object-Relational Mapping*) dla .NET. EF Core umożliwia mapowanie obiektów C# na tabele bazodanowe, co przyspiesza i ułatwia zarządzanie danymi w aplikacji. Przykładem jest poniższy listing przedstawiający klasę `ApiDbContext` która korzysta z funkcji konfiguracji baz danych Entity Framework Core. Klasa ta rozszerza `IdentityDbContext`, dostarczając kontekst bazy danych dla aplikacji z uwzględnieniem tożsamości użytkowników oraz dodatkowych encji.

Listing 36 przedstawia przykład klasy `ApiDbContext`, która korzysta z Entity Framework Core do konfiguracji bazy danych w aplikacji.

Listing 36 Przykład zastosowania Entity Framework Core. Źródło: Opracowanie własne

```
    public class ApiDbContext(
          DbContextOptions<ApiDbContext> options,
          IConfiguration configuration,
          UserIdService userIdService
          ) : IdentityDbContext<User, IdentityRole<Guid>, Guid>(options)
    {
          private readonly Guid? _userId = userIdService.GetUserId();

          public DbSet<FileUpload> FileUploads { get; init; } = null!;

          public DbSet<Restaurant> Restaurants { get; init; } = null!;

          // ... Reszta setów

          protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
          {
          var connString = configuration.GetConnectionString("Default");

          optionsBuilder.UseSqlServer(
                    connString ?? throw new InvalidOperationException("Connection string
'Default' not found"),
                    x => x.UseNetTopologySuite());
          }
```

```
        protected override void OnModelCreating(ModelBuilder builder)
        {
        base.OnModelCreating(builder);

          builder.ApplyConfigurationsFromAssembly(typeof(ApiDbContext).Assembly);

        builder.Entity<Notification>()
                 .HasQueryFilter(n => n.TargetUserId == _userId);
        }
    }
```

Dzięki zastosowanym rozwiązaniom aplikacja Reservant może efektywnie i bezpiecznie obsługiwać operacje CRUD (ang. *Create, Read, Update, Delete*) dla użytkowników, restauracji i rezerwacji, integrując różne elementy systemu w jednolitym modelu danych.

5.3.4 Opis użycia bibliotek

Aplikacja serwerowa omawianego systemu jest efektem złożonej implementacji wielu bibliotek oraz zależności. Poniższe podrozdziały omawiają szczegółowo w jaki sposób zdecydowano się użyć wspomnianych w rozdziale 4 narzędzi.

5.3.4.1 FirebaseAdmin

Firebase Admin SDK to zbiór bibliotek serwera umożliwiający interakcję Firebase ze środowisk o podwyższonych uprawnieniach do wykonywania działań takich jak zapytania i mutacje w usłudze Firebase Data Connect na potrzeby zbiorczego zarządzania danymi oraz innych operacji o podwyższonych uprawnieniach. Niniejsza praca wykorzystuje to rozwiązanie w celu wysyłania powiadomień push do użytkowników.

Listing 37 przedstawia przykład implementacji metody wysyłania powiadomienia push do konkretnego użytkownika przy użyciu Firebase Admin SDK.

Listing 37 Przykład użycia wysyłanie powiadomienia push. Źródło: Opracowanie własne

```
    /// <summary>
    /// Send a push notification to a specific user
    /// </summary>
    public async Task SendNotification(Notification notification)
    {
          if (FirebaseApp.DefaultInstance is null)
          {
          return;
          }

          var targetUser = await context.FindAsync<User>(notification.TargetUserId)
          ?? throw new InvalidOperationException("Target user not found");
          if (targetUser.FirebaseDeviceToken is null)
          {
          return;
          }

          var userCulture = targetUser.Language;

          try
          {
          await FirebaseMessaging.DefaultInstance.SendAsync(new FirebaseMessage
          {
```

```
                Notification = ComposeNotification(notification, userCulture),
                   Token = targetUser.FirebaseDeviceToken,
            });
            }
            catch (FirebaseException ex)
            {
              logger.FirebaseMessagingError(ex, notification.TargetUserId);
            }
    }
```

5.3.4.2 SmartFormat

SmartFormat to lekka biblioteka szablonów tekstowych napisana w języku C#, która może być zamiennikiem dla „string.Format" . Ponadto SmartFormat może formatować dane za pomocą nazwanych symboli zastępczych, list, lokalizacji, pluralizacji i innych inteligentnych rozszerzeń.

W implementacji serwera została zastosowana w celu formatowania tytułów i opisów powiadomień push do użytkowników. Prezentowana poprzez Listing 38 metoda, na podstawie danych z obiektu powiadomienia (obiekt `Notification`), wybiera szablon z zasobów aplikacji w wybranym języku (`CultureInfo culture`) i wypełnia ten szablon danymi z obiektu powiadomienia.

Listing 38 Przykład tłumaczenia powiadomień na wybrany język. Źródło: Opracowanie własne

```
    /// <summary>
    /// Compose human-readable notification from a Notification object
    /// </summary>
    /// <param name="notification">The Notification object</param>
    /// <param name="culture">Language of the notification</param>
    private FirebaseNotification ComposeNotification(Notification notification, CultureInfo
culture)
    {
          var notificationType = notification.Details.GetType().Name;

          var originalCulture = CultureInfo.CurrentUICulture;
          CultureInfo.CurrentUICulture = culture;

          var title = Smart.Format(localizer[$"{notificationType}.Title"],
notification.Details);
          var body = Smart.Format(localizer[$"{notificationType}.Details"],
notification.Details);

          CultureInfo.CurrentUICulture = originalCulture;

          return new FirebaseNotification
          {
          Title = title,
          Body = body,
          ImageUrl = urlService.GetUrlForFileName(notification.PhotoFileName)?.ToString(),
          };
    }
```

5.3.4.3 FluentValidation

FluentValidation jest biblioteką, która pozwala pisać reguły walidacyjne za pomocą interfejsu fluent. Taki sposób pisania kodu jest o wiele bardziej zrozumiały dla czytelnika i upraszcza utrzymywanie kodu. Dzięki temu, nawet przy skomplikowanej logice walidacyjnej i wielu krzyżujących się walidacjach, wciąż będzie można nadążyć za tym, co się dzieje w aplikacji. Dodatkowo biblioteka znakomicie wspiera wstrzykiwanie zależności, co upraszcza testowanie

walidatorów i reguł walidacji. Całość w prosty sposób wpina się w aktualnie używany framework ASP.NET.

Listing 39 pokazuje zdefiniowane reguły, które muszą zostać spełnione dla obiektów w aplikacji. W pierwszym fragmencie kodu został pokazany przykład implementacji dla obiektu menu. W kolejnych fragmentach zaprezentowano uruchomienie sprawdzenia reguł walidacyjnych.

Listing 39 Walidacja danych i zdefiniowanie poprawnego wzorca. Źródło: Opracowanie własne

```
//zdefiniowanie reguł po których sprawdzamy poprawność danych
/// <summary>
/// Validator for <see cref="Menu"/>
/// </summary>
public class MenuValidator : AbstractValidator<Menu>
{
      /// <inheritdoc />
      public MenuValidator(ApiDbContext context)
      {
      RuleFor(x => x.MenuType)
            .IsInEnum();

      RuleFor(x => x.RestaurantId)
            .RestaurantExists(context);

      RuleFor(x => x.DateUntil)
            .DateInFuture();
      }
}
//funkcja uruchamiająca sprawdzenie i zwracająca ewentualne błędy
public async Task<ValidationResult> ValidateAsync<T>(T instance, Guid? userId)
{
      var validationContext = new ValidationContext<T>(instance);
      validationContext.RootContextData["UserId"] = userId;
      return await serviceProvider
               .GetRequiredService<IValidator<T>>()
               .ValidateAsync(validationContext);
}

//użycie w kodzie
var result = await validationService.ValidateAsync(menu, user.Id);
if (!result.IsValid)
{
      return result;
}
```

5.3.4.4 ASP.NET Core Identity

ASP.NET Core Identity to interfejs API, który obsługuje funkcje logowania interfejsu użytkownika aplikacji Reservant.

Umożliwia zarządzanie użytkownikami, hasłami, danymi profilu, rolami, oświadczeniami, tokenami, potwierdzeniem wiadomości e-mail i nie tylko.

Użytkownicy mogą utworzyć konto przy użyciu informacji logowania przechowywanych w Identity programie lub użyć zewnętrznego dostawcy logowania. Obsługiwani zewnętrzni dostawcy logowania to Facebook, Google, Konto Microsoft i Twitter.

Identity Program jest skonfigurowany do MS SQL do przechowywania nazw użytkowników, haseł i danych profilu.

Autentykacja jest zrobiona w aplikacji za pomocą tokenów JWT, które zawierają w sobie dane o użytkowniku razem z wygenerowanym przez aplikację podpisem uwierzytelniającym zapewniającym poprawność danych. Listing 40 definiuje sposób sprawdzenia wygenerowanego tokena JWT, a Listing 41 ilustruje obsługę przechowywania użytkowników w bazie danych.

Listing 40 Konfiguracja autentyfikacji za pomocą JWT. Źródło: Opracowanie własne

```
    services
        .AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
          .AddJwtBearer(o =>
          {
          var jwtOptions = configuration.GetSection(JwtOptions.ConfigSection)
                .Get<JwtOptions>() ?? throw new InvalidOperationException("Failed to read
JwtOptions");

            o.TokenValidationParameters = new TokenValidationParameters
          {
                ValidIssuer = jwtOptions.Issuer,
                ValidAudience = jwtOptions.Audience,
                IssuerSigningKey = new SymmetricSecurityKey(jwtOptions.GetKeyBytes()),
                ValidateIssuerSigningKey = true
          };

          o.Events = new JwtBearerEvents();
            o.Events.OnTokenValidated += context =>
          {
                context.HttpContext.Items.Add(
                    HttpContextItems.AuthExpiresUtc, context.Properties.ExpiresUtc);
                  return Task.CompletedTask;
          };
          })
```

Listing 41 Konfiguracja modeli użytkownika i roli. Źródło: Opracowanie własne

```
    services.AddAuthorization();
    services
        .AddIdentityCore<User>(o =>
          {
            o.SignIn.RequireConfirmedEmail = false;
            o.SignIn.RequireConfirmedPhoneNumber = false;
          })
        .AddRoles<IdentityRole<Guid>>()
        .AddEntityFrameworkStores<ApiDbContext>() // Przechowujemy w bazie danych aplikacji
        .AddDefaultTokenProviders();
```

Listing 42 prezentuje funkcjonalność, która jest dostępna jedynie dla użytkowników z przypisaną wybraną rolą. W omawianym przykładzie jest to funkcjonalność usuwania menu dostępna tylko dla właściciela restauracji.

Listing 42 Przykład weryfikacji zalogowanego użytkownika. Źródło: Opracowanie własne

```
    [HttpDelete("{menuId:int}")]
    [Authorize(Roles = Roles.RestaurantOwner)] // <- Końcówka dostępna tylko dla
właściciela restauracji
    public async Task<ActionResult> DeleteMenu(int menuId){
          var user = await userManager.GetUserAsync(User); // <- Pobieramy obiekt
zalogowanego użytkownika
          if (user is null){
          return Unauthorized();
          }
          var res = await service.DeleteMenuAsync(menuId, user);
          return OkOrErrors(res);
    }
```

5.3.4.5 Microsoft.AspNetCore.OpenApi

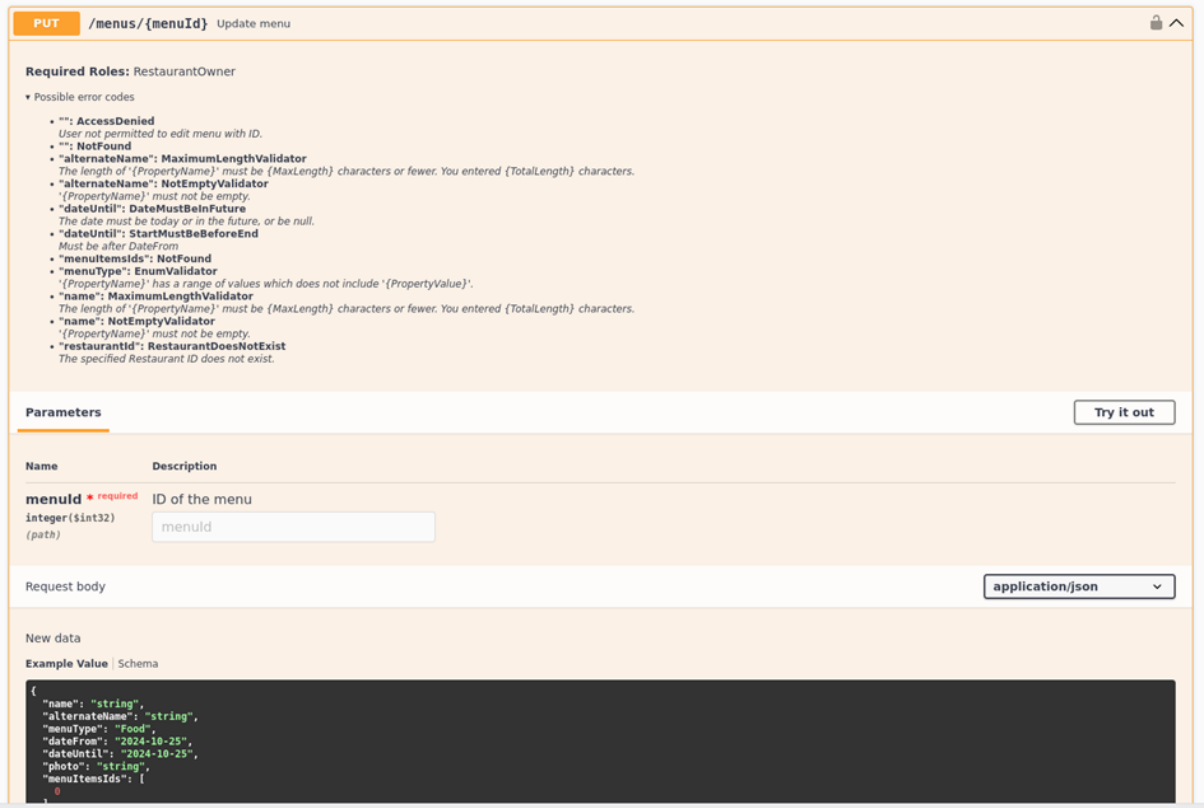
ASP.NET Core udostępnia pakiet do interakcji ze specyfikacjami interfejsu Microsoft.AspNetCore.OpenApi OpenAPI dla punktów końcowych. Pakiet działa jako link między modelami OpenAPI zdefiniowanymi w pakiecie i punktami końcowymi zdefiniowanymi w Microsoft.AspNetCore.OpenApi minimalnych interfejsach API. Pakiet udostępnia interfejs API, który analizuje parametry, odpowiedzi i metadane punktu końcowego w celu konstruowania typu adnotacji interfejsu OpenAPI używanego do opisywania punktu końcowego.

W praktyce oznacza to, że biblioteka pozwala na stworzenie dokumentu, który następnie przekazywany jest do Swaggera, Swagger czyta ten plik i wyświetla odpowiednio końcówki.

Listing 43 przedstawia kod źródłowy funkcjonalności aktualizacji menu, który definiuje odpowiednie atrybuty widoczne jako opis w aplikacji Swagger (Rysunek 54).

Listing 43 Przykład końcówki z dokumentacją w kodzie. Źródło: Opracowanie własne

```
    /// <summary>
    /// Update menu
    /// </summary>
    /// <param name="request">New data</param>
    /// <param name="menuId">ID of the menu</param>
    /// <returns></returns>
    [HttpPut("{menuId:int}")]
    [Authorize(Roles = Roles.RestaurantOwner)]
    [ProducesResponseType(200)]
    [ProducesResponseType(400)]
    [MethodErrorCodes<RestaurantMenuService>(nameof(RestaurantMenuService.UpdateMenuAsync))
]
    public async Task<ActionResult<MenuVM>> UpdateMenu(UpdateMenuRequest request, int
menuId)
    {
          ...
    }
```



Rysunek 54 Wygląd końcówki w Swaggerze. Źródło: Opracowanie własne

5.3.4.6 NetTopologySuite

NetTopologySuite (NTS) to biblioteka przestrzenna platformy .NET. Program EF Core umożliwia mapowanie na typy danych przestrzennych w bazie danych przy użyciu typów NTS w modelu. W niniejszej pracy biblioteka stanowi wsparcie dla typów geograficznych EF. Przechowywane są punkty określające położenie danej restauracji co umożliwia określenie odległości restauracji od użytkownika. Listing 44 przedstawia przykład kodu działającego na danych geograficznych. Fragment ten prezentuje deklarację punktu lokalizacji restauracji na mapie oraz wyszukanie restauracji znajdujących się najbliżej użytkownika.

Listing 44 Przykład zastosowania NTS. Źródło: Opracowanie własne

```
//Pole lokalizacji w klasie restauracji:
public required Point Location { get; set; }

//Wyszukiwanie restauracji najbliższych do użytkownika:
origin = geometryFactory.CreatePoint(new Coordinate(origLon!.Value, origLat!.Value));
query = query.OrderBy(r => origin.Distance(r.Location));
```

5.3.5 Implementacja autorskich bibliotek

W niektórych sytuacjach zespół implementacji serwera potrzebował rozwiązań, które były wyjątkowe dla specyfiki wdrożenia. W celu ułatwienia sobie pracy utworzono autorskie biblioteki. Poniższe podrozdziały skupiają się na opisie wspomnianych bibliotek.

5.3.5.1 ErrorCodeDocs

ErrorCodeDocs jest autorską biblioteką zespołu implementacji serwera, pozwalająca na łatwe udokumentowanie kodów błędów, które mogą zostać zwrócone przez końcówkę lub metodę.

Głównym powodem utworzenia biblioteki było napotkanie problemu, który dotyczył braku dokumentacji możliwych kodów błędów sygnalizujących niepowodzenie operacji. Kody błędów trzeba byłoby określać ręcznie co mogłoby powodować liczne pomyłki i wymagać szczególnej uwagi przy wprowadzaniu zmian w kodzie. Dokumentacja kodów błędów była niezbędna dla zespołów implementacji pracujących nad częścią kliencką aplikacji i aplikacją mobilną.

Podstawową funkcją biblioteki jest umożliwienie udokumentowania możliwych do wystąpienia kodów błędów, tuż przy kodzie sprawdzającym wystąpienie błędu. Dodatkowo zaprezentowane rozwiązanie pozwala na zminimalizowanie powtórzeń opisów błędów, umożliwiając odniesienie się do opisanych uprzednio kawałków kodu, odpowiadających za konkretne kody błędów. Na podstawie wspomnianych opisów błędów, biblioteka dodaje do interfejsu Swagger odpowiednie informacje o kodach błędów. Listing 45 przedstawia przykład opisu kodów błędów za pomocą omawianej biblioteki.

Listing 45 Przykład opisu kodów błędów. Źródło: Opracowanie własne

```
    [MethodErrorCodes<RestaurantService>(nameof(RestaurantService.CreateRestaurantAsync))]
    public async Task<ActionResult<MyRestaurantVM>>
CreateRestaurant(CreateRestaurantRequest request)
    {
          var user = await userManager.GetUserAsync(User);
          if (user is null)
          {
          return Unauthorized();
          }

          var result = await restaurantService.CreateRestaurantAsync(request, user);
          return OkOrErrors(result);
    }

    [ErrorCode(nameof(CreateRestaurantRequest.GroupId), ErrorCodes.NotFound)] // <- Metoda
może zwrócić kod NotFound
    [ErrorCode(nameof(CreateRestaurantRequest.GroupId), ErrorCodes.AccessDenied,
          "Group with ID is not owned by the current user")] // <- Może zwrócić kod
AccessDenied (z opisem co to oznacza)
    [ValidatorErrorCodes<CreateRestaurantRequest>] // <- Waliduje wewnątrz
CreateRestaurantRequest i zwraca błędy
    [ValidatorErrorCodes<Restaurant>] // <- Waliduje wewnątrz Restaurant i zwraca błędy
    public async Task<Result<MyRestaurantVM>> CreateRestaurantAsync(CreateRestaurantRequest
request, User user)
    {
          ...
    }
```

5.3.5.2 LogsViewer

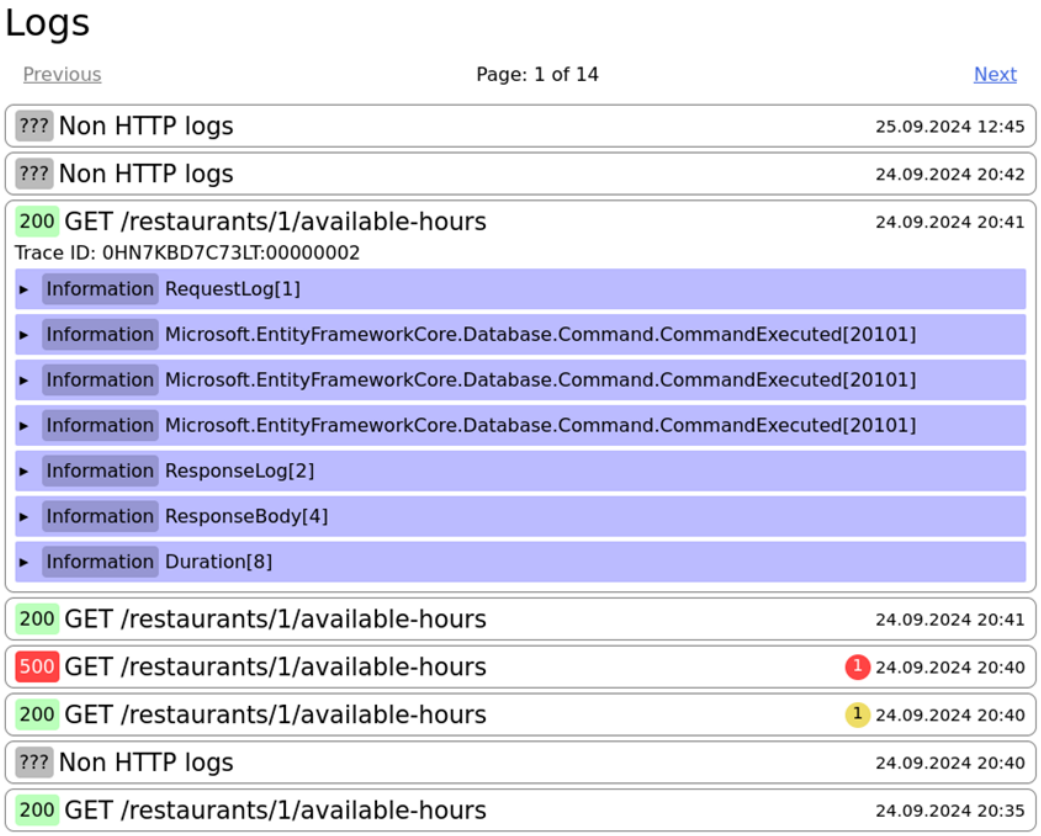
LogsViewer to stworzona przez zespół implementacji serwera, autorska biblioteka, służąca do zapisywania logów do bazy danych SQL LITE.

Biblioteka została stworzona w celu przedstawienia logów aplikacji w sposób przejrzysty i przyjemny dla oka oraz w celu ułatwienia dostępu do logów. Przed wdrożeniem wspomnianego rozwiązania, aby odczytać dane z logów należało połączyć się z serwerem za pomocą terminala a następnie pobrać logi w formacie tekstowym. Zastosowana biblioteka automatycznie pobiera logi i przedstawia je na stronie w poniżej zaprezentowanym formacie.

Rysunek 55 przedstawia interfejs graficzny, który jest efektem implementacji biblioteki. Logi w tym interfejsie pogrupowane są względem akcji, do których się odnoszą. Dla każdej akcji została zaprezentowana jej nazwa, czas wywołania oraz wynik końcowy po wywołaniu akcji. Listing 46 pokazuje sposób użycia tej biblioteki w aplikacji, rejestrując jej usługę i włączając interfejs użytkownika dla przeglądania logów.

Listing 46 Użycie systemu logowania w Aplikacji. Źródło: Opracowanie własne

```
app.Services.RegisterLogsViewerProvider();
app.UseLogsViewerUI();
```



Rysunek 55 Sposób wyświetlania LogsViewer. Źródło: Opracowanie własne

Automatyzacja dokumentacji i logów znacznie zwiększyła przejrzystość kodu, pozwoliła na łatwiejsze śledzenie zmian oraz zapewniła szybką dostępność informacji o błędach i funkcjonalnościach aplikacji.

5.3.6 Websockety

WebSocket pozwala użytkownikowi wysyłać i odbierać wiadomości na serwerze. Zasadniczo jest to sposób komunikacji między klientem a serwerem. WebSocket stanowi dużo lepsze rozwiązanie pod względem wydajnościowym niż Polling i Long Polling, ponieważ nie wymaga wysyłania żądania w celu udzielenia odpowiedzi. WebSockety polegają na tym, że klient „nasłuchuje" serwera, który wyśle klientowi wiadomość, gdy będzie ona gotowa. Listing 47 pokazuje metodę wysyłającą powiadomienia za pomocą WebSocketów.

Listing 47 Metoda wysyłająca powiadomienia websocket. Źródło: Opracowanie własne

```
/// <summary>
/// Send messages from PushService in a loop
/// </summary>
private static async Task PushMessages(
      PushService source, WebSocket socket, Guid userId, CancellationToken cancelToken)
{
      var messages = new ConcurrentQueue<byte[]>();

    source.Subscribe(userId, EnqueueAuthorizedMessages);
      try
      {
      while (!cancelToken.IsCancellationRequested)
      {
                if (!messages.TryDequeue(out var message))
                {
                await Task.Delay(MessagePollDelayMs, cancelToken);
                 continue;
                }

                await socket.SendAsync(
                 message,
                 WebSocketMessageType.Text,
                 endOfMessage: true,
                 cancelToken);
      }
      }
      catch (OperationCanceledException) { }
      finally
      {
        source.Unsubscribe(userId, EnqueueAuthorizedMessages);
      }

      return;

      void EnqueueAuthorizedMessages(byte[] message)
      {
        messages.Enqueue(message);
      }
}
```

5.4 Implementacja klienckiej aplikacji webowej

Do stworzenia aplikacji webowej zastosowano bibliotekę React wraz z TypeScript, co umożliwiło budowę dynamicznego interfejsu użytkownika z zachowaniem bezpieczeństwa typowania. Stylizację aplikacji oparto na HTML i CSS, uzupełniając je o biblioteki Material-UI oraz Tailwind CSS. Wykorzystanie tych bibliotek pozwoliło na spójną definicję komponentów oraz ich estetyczną personalizację, co pozytywnie wpłynęło na czytelność i spójność interfejsu.

Do zarządzania formularzami oraz ich walidacji zaimportowano biblioteki Formik i Yup, co usprawniło proces obsługi danych wprowadzanych przez użytkownika oraz zapewniło poprawność ich walidacji. W razie konieczności dodawano dodatkowe biblioteki, aby dostosować funkcjonalność aplikacji do wymagań projektowych. Praca nad kodem odbywała się w środowisku Visual Studio Code, które oferuje szeroki zakres narzędzi wspierających proces tworzenia aplikacji webowych.

5.4.1 Struktura projektu aplikacji webowej

Struktura plików w projekcie została zorganizowana w sposób ułatwiający nawigację i zarządzanie zasobami aplikacji. Poniżej znajduje się szczegółowy opis kluczowych folderów i plików:

- Folder *public* – zawiera zasoby publicznie dostępne, a jednym z najważniejszych plików w tym folderze jest index.html, który stanowi punkt wejścia aplikacji. Jest to główny plik HTML, do którego podpinane są skrypty JavaScript generujące interfejs użytkownika.
- Folder *src* – przechowuje wszystkie istotne zasoby aplikacji;
    - Routing – zarządza nawigacją pomiędzy stronami. Cały routing został zdefiniowany w pliku *src/App/App.tsx* w postaci drzewa. Listing 48 przedstawia istotne elementy przy tworzeniu routingu aplikacji. Listing 48 przedstawia istotne elementy przy tworzeniu routingu aplikacji,

Listing 48 Fragment kodu App.tsx. Źródło: Opracowanie własne

```
const router = createBrowserRouter([
 {
   path: '/',
   element: <LandingPage />
 },
 {
   path: 'reservant',
   element: <Root />,
   loader: checkAuthLoader,
   children: [
     {
       path: 'home',
       element: <HomePage />
     },
     /* ... pozostałe ścieżki zagnieżdżone ... */
   ]
 },
 {
   path: 'login',
   element: <Login />,
   loader: redirectIfLoggedIn
 },
 /* ... reszta konfiguracji tras ... */
]);
```

- Czcionki i obrazki – W podkatalogu *asstes* zostały umieszczone pliki multimedialne wykorzystywane w aplikacji,
- Komponenty – w podkatalogu *components* umieszczono modułowe elementy interfejsu aplikacji. Jest to największa część struktury projektu, ponieważ zawiera wszystkie elementy, które są bezpośrednio widoczne dla użytkownika. W *Components* znajdują się zarówno podstawowe komponenty, jak przyciski czy pola formularzy, jak i bardziej złożone moduły, takie jak całe sekcje czy widoki aplikacji. Każdy komponent odpowiada za konkretny fragment interfejsu, co pozwala na ich łatwe ponowne użycie i utrzymanie przejrzystego, modularnego kodu,
- Konteksty – W podkatalogu *contexts* zostały zdefiniowane globalne stany dostępne w różnych częściach aplikacji,
- Hooki – W podkatalogu *hooks* zostały umieszone niestandardowe hooki, które rozszerzają funkcjonalność komponentów. Jednym z takich hooków w naszej aplikacji jest `useValidationSchemas`, który za pomocą biblioteki yup pozwala walidować formularze np. rejestracji użytkowników lub logowania,
- Enumy - W pliku *src/services/enum.tsx* zostały umieszone wszystkie enumeracje, wykorzystywane w różnych miejscach w aplikacji,
- Błędy – W pliku *src/services/Erros.tsx* zostały umieszone niestandardowe typy błędów występujące w aplikacji,
- Funkcje API – W pliku *src/services/APIconn.tsx* znajdują się funkcje odpowiedzialne za komunikację z API, które są wykorzystywane w większości komponentów aplikacji. Funkcje te mają wstępnie zdefiniowaną ścieżkę serwera oraz wbudowaną obsługę błędów, co pozwala na znaczne uproszczenie kodu w poszczególnych komponentach. Przykładowo funkcja `fetchGET`, którą pokazuje Listing 49 jest oznaczona jako async, co pozwala używać wewnątrz niej słowa kluczowego await. Dzięki temu możemy "czekać" na zakończenie operacji, takich jak pobranie tokena czy odpowiedzi z serwera, zanim przejdziemy do kolejnych kroków w kodzie. `Cookies.get("token")` służy do pobrania wartości tokena z ciasteczek, który jest wykorzystywany do autoryzacji użytkownika w systemie. Token ten zostaje dodany w nagłówku zapytania HTTP, co umożliwia serwerowi weryfikację, czy użytkownik ma prawo dostępu do danych. `await response.json()` czeka na zakończenie procesu konwersji odpowiedzi z serwera do formatu JSON, który jest standardowym sposobem przesyłania danych między aplikacjami, ułatwiając ich dalsze przetwarzanie. `await` sprawia, że kod "czeka" na pełne przetworzenie odpowiedzi przed przejściem do kolejnych operacji, zapewniając, że dane będą dostępne w momencie ich użycia. Listing 50 przedstawia użycie tej metody w innym komponencie;

Listing 49 Implementacja funkcji fetchGET. Źródło: Opracowanie własne

```
export const fetchGET = async (connString: string) => {
 const token = Cookies.get("token");


 const response = await fetch(
   `${process.env.REACT_APP_SERVER_IP}${connString}`,
   {
     headers: {
       Authorization: `Bearer ${token as string}`,
     },
   },
 );
```

```
  if (!response.ok) {
    const errorData = await response.json();
    throw new FetchError("" , errorData.status, errorData.errors);
  }


  const data = await response.json();


  return data;
};
```

Listing 50 Wykorzystanie funkcji fethGET w innym komponencie. Źródło: Opracowanie własne

```
      const response = await fetchGET("/user/employees");
```

- Typy (ang. *Types*) – W pliku *src/services/types.tsx* znajdują się definicje typów dla TypeScript, które poprawiają czytelność kodu oraz zapewniają ścisłą kontrolę typów w całej aplikacji, zwiększając jej bezpieczeństwo i stabilność,
- Tłumaczenia – W podkatalogu translations zostały umieszczone dwa podkatalogi en i pl oba zawierające pliki *global.json*, który w postaci drzewa przypisują słowa w odpowiednim języku do tej samej zmiennej. Listing 51 oraz Listing 52 przedstawiają fragmenty plików global.json,

Listing 51 Fragment en/global.json. Źródło: Opracowanie własne

```
"add-employee": {
      "isBackdoorEmployee": "Backdoor employee",
      "isHallEmployee": "Hall employee" ,
      "addEmployee": "Assign employee"
  },
```

Listing 52 Fragment pl/global.json. Źródło: Opracowanie własne

```
"add-employee": {
      "isBackdoorEmployee": "Pracownik zaplecza",
      "isHallEmployee": "Parcownik sali" ,
      "addEmployee": "Dodaj zatrudnienie"
  },
```

- Pliki główne projektu (poza folderami public i src);
  - *package.json* oraz *package-lock.json* – przechowują informacje o zależnościach projektu, wersjach bibliotek tj. MUI czy tailwind oraz skrypty uruchamiane podczas budowania i uruchamiania aplikacji,
  - *tailwind.config.js* – konfiguracja frameworka Tailwind CSS, pozwalająca dostosować stylizację aplikacji,

- *Jenkinsfile* – skrypt konfiguracyjny dla narzędzia Jenkins, używany do automatyzacji procesu budowania, testowania i wdrażania aplikacji,
- *Dockerfile* – plik definiujący konfigurację obrazu Docker, umożliwiający łatwe tworzenie kontenerów do uruchamiania aplikacji w spójnych środowiskach produkcyjnych.

5.4.2 Opis użycia bibliotek

Napisanie aplikacji webowej charakteryzuje się w wielu przypadkach koniecznością użycia dużej liczby bibliotek pomocniczych. Szczególnie dotyczy to przypadku omawianej technologii implementacji jaką jest React.js. Fragment ten opisuje sposób implementacji tych bibliotek w prezentowanym systemie.

5.4.2.1 Material UI

W projekcie biblioteka Material-UI (MUI) została wykorzystana głównie do tworzenia i stylistyki komponentów interfejsu użytkownika. Dzięki MUI, udało się zaimplementować spójny wygląd aplikacji oraz użyteczne komponenty, które poprawiają doświadczenie użytkownika i przyspieszają proces tworzenia. Najczęściej w projekcie korzystaliśmy z tej biblioteki przy tworzeniu tabel, ekranów ładowania, przycisków, dialogów, wstawianiu ikon i wiele innych przypadkach.

Listing 53 ilustruje użycie komponentów Material-UI do stworzenia dialogu potwierdzenia akcji. W tym przykładzie komponent `ConfirmationDialog` wykorzystuje takie elementy MUI jak `Dialog`, `DialogActions`, `DialogContent`, `DialogTitle` oraz `Button`, aby zapewnić intuicyjny i elegancki sposób obsługi decyzji użytkownika.

Listing 53 Użycie komponentów MUI. Źródło: Opracowanie własne

```
import React, { ReactNode } from "react";
import {
 Button,
 Dialog,
 DialogActions,
 DialogContent,
 DialogTitle,
} from "@mui/material";
import { useTranslation } from "react-i18next";
interface ConfirmationDialogProps {
 open: boolean;
 onClose: () => void;
 onConfirm: () => void;
 confirmationText: string;
 onAlt?: () => void;
 altText?: string;
}

const ConfirmationDialog: React.FC<ConfirmationDialogProps> = ({
 open,
 onClose,
 onConfirm,
 onAlt,
 altText,
 confirmationText,
}) => {
 const [t] = useTranslation("global");
 const handleConfirm = () => {
   onConfirm();
   onClose();
 };
```

```
  return (
    <Dialog open={open} onClose={onClose}>
      <DialogTitle>{t("general.confirmation")}</DialogTitle>
      <DialogContent>{confirmationText}</DialogContent>
      <DialogActions>
        {(onAlt || altText) && (
          <Button onClick={onAlt} color="error">
            {altText}
          </Button>
        )}
        <Button onClick={onClose} color="primary">
          {t("general.cancel")}
        </Button>
        <Button onClick={handleConfirm} color="error">
          {t("general.yes")}
        </Button>
      </DialogActions>
    </Dialog>
  );
};
export default ConfirmationDialog;
```

5.4.2.2 Tailwind

W projekcie Tailwind CSS pozwolił na szybkie prototypowanie, łatwe dostosowanie stylów do różnych potrzeb, zapewnienie responsywności aplikacji oraz na łatwą personalizację interfejsu użytkownika. Dzięki niemu zminimalizowano ilość skomplikowanych plików CSS oraz zdefiniowano własne kolory w pliku tailwind.config.js co pozwala na łatwą zmianę koloru w całej aplikacji.

5.4.2.3 Formik

W projekcie biblioteka Formik została w szczególności wykorzystana do uproszczenia zarządzania formularzami rejestracji i logowania użytkownika. Dzięki niej, obsługa stanu formularzy, walidacja danych oraz zarządzanie błędami stały się bardziej efektywne i mniej czasochłonne. Formik pozwolił na centralne zarządzanie wartościami formularzy, co zminimalizowało potrzebę ręcznego pisania kodu do obsługi wprowadzanych danych, a tym samym poprawiło czytelność i utrzymanie kodu.

Kluczowe elementy formika, które przedstawia Listing 54 to m.in. `initialValues` - określa początkowe wartości pól formularza (np. login, hasło), `validationSchema` – zawiera schemat walidacji, który jest odpowiedzialny za sprawdzenie poprawności danych wpisanych przez użytkownika, `onSubmit` – funkcja, która zostanie wywołana po poprawnym wysłaniu formularza. Komponent Formik, który obsługuje generowanie formularza. Każde pole w formularzu jest opakowane w komponent `Field`, który integruje się z biblioteką Formik. W przypadku `TextField`, jest to komponent z biblioteki Material-UI, który renderuje pole wejściowe z odpowiednimi atrybutami.

Listing 54 Komponent Login wykorzystujący Formik. Źródło: Opracowanie własne

```
import { Formik, Form, Field, FormikValues } from "formik";
import { useTranslation } from "react-i18next";
import { TextField } from "@mui/material";
import { Visibility, VisibilityOff } from "@mui/icons-material";
import { useValidationSchemas } from "../../hooks/useValidationSchema";

const initialValues = {
 login: "",
 password: "",
};

const Login: React.FC = () => {
```

```
    const [t] = useTranslation("global");
    const { loginSchema } = useValidationSchemas();
    const [showPassword, setShowPassword] = useState<boolean>(false);

    // Placeholder for form submission logic
    const onSubmit = async (
      values: FormikValues,
      { setSubmitting }: { setSubmitting: (isSubmitting: boolean) => void }
    ) => {
      // logika logowania (np. fetch i zapis do cookies)
      // wywołanie API
      setSubmitting(false);
    };

    return (
      <div className="h-full w-full bg-[url('/src/assets/images/bg.png')] bg-cover">
        <div className="login-gradient flex h-full w-full items-center justify-center bg-
opacity-20">
          <div className="flex h-[500px] w-[900px] items-center rounded-lg border-2
border-white bg-black bg-opacity-60 shadow-2xl backdrop-blur-md">
            <div className="flex h-full w-2/3 flex-col items-center justify-center gap-12
rounded-r-lg">
              <Formik
                initialValues={initialValues}
                validationSchema={loginSchema}
                onSubmit={onSubmit}
              >
                {(formik) => (
                  <Form className="w-full">
                    <div className="form-container flex h-full flex-col items-center gap-
8">
                      <div className="flex w-full flex-col items-center gap-6">
                        <Field
                          type="text"
                          id="login"
                          name="login"
                          label="LOGIN"
                          variant="standard"
                          as={TextField}
                        />
                        <div className="relative w-4/5">
                          <Field
                            id="password"
                            name="password"
                            type={showPassword ? "text" : "password"}
                            label="PASSWORD"
                            variant="standard"
                            as={TextField}
                          />
                          <span
                            className="absolute right-[0%] top-[40%] cursor-pointer text-
white"
                            onClick={() => setShowPassword(!showPassword)}
                          >
                            {showPassword ? <VisibilityOff /> : <Visibility />}
                          </span>
                        </div>
                      </div>

                      <button
                        type="submit"
                        disabled={!formik.isValid}
                        className={`flex h-[50px] w-4/5 items-center justify-center
rounded-lg shadow-md ${formik.isValid ? "bg-primary text-white" : "bg-grey-1"}`}
                      >
                        LOGIN
```

```
                    </button>
                  </div>
                </Form>
              )}
            </Formik>
          </div>
        </div>
      </div>
    </div>
  );
};

export default Login;
```

5.4.2.4 Yup

Biblioteka Yup jest używana do walidacji danych formularzy w połączeniu z Formikiem, co znacząco ułatwia kontrolowanie poprawności danych wprowadzanych przez użytkowników. Przykładowo, dzięki Yup, można zdefiniować zasady, które muszą zostać spełnione przed wysłaniem formularza, co zapewnia, że dane są poprawne i spełniają wymagania aplikacji. Listing 55 prezentuje różne metody walidacji, takie jak `string`(), `array`(), `mixed`(), `min`(), `max`(), i `required`(), służą do określenia, jakie zasady muszą spełniać dane wejściowe.

Listing 55 Wykorzystanie biblioteki yup w schemacie. Źródło: Opracowanie własne

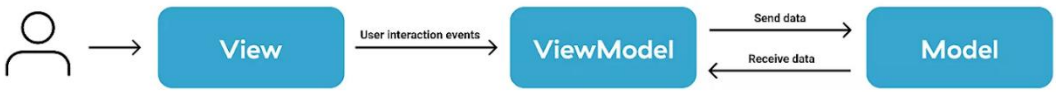
```
const RestaurantRegisterStep2Schema = yup.object({
  description: yup
    .string()
    .max(200, t("errors.restaurant-register.description.max"))
    .min(3, t("errors.restaurant-register.description.min"))
    .required(t("errors.restaurant-register.description.required")),
  tags: yup.array().min(3, t("errors.restaurant-register.tags.min")),
  logo: yup.mixed().required(t("errors.restaurant-register.logo.required")),
  photos: yup
    .mixed()
    .required(t("errors.restaurant-register.photos.required")),
});
```

5.5 Implementacja aplikacji mobilnej

Telefony komórkowe zaczęły odgrywać w ludzkich życiach istotną rolę. Ogromna część życia społecznego odbywa się właśnie tam. Ze względu na to, że omawiany system jest nakierowany m.in na bezpośrednie interakcje międzyludzkie, kluczowe było zaimplementowanie aplikacji mobilnej w sposób pozwalający użytkownikom na intuicyjne poruszanie się po niej. Do tego celu użyto nowoczesnych technologii oraz praktyk, które pozwolą stworzyć świeży wygląd w połączeniu z wysoką wydajnością.

5.5.1 Omówienie wzorca MVVM

MVVM (ang. *Model-View-ViewModel*) to wzorzec architektoniczny stosowany w aplikacjach mobilnych, który oddziela logikę biznesową od interfejsu użytkownika. Składa się on z trzech komponentów: Modelu, który przechowuje dane i logikę biznesową; Widoku, odpowiedzialnego za prezentację interfejsu; oraz ViewModelu, który pośredniczy między modelem a widokiem. *ViewModel* zarządza danymi i stanem widoku oraz przetwarza interakcje użytkownika, zapewniając dwukierunkowe połączenie danych. Dzięki MVVM kod jest bardziej modułowy, łatwiejszy do testowania i utrzymania, co sprzyja budowaniu wydajnych i skalowalnych aplikacji mobilnych. Diagram ilustrujący działanie wzorca MVVM przedstawia Rysunek 56.



Rysunek 56 Diagram ilustrujący działanie wzorca MVVM. Źródło: [64]

W omawianym systemie zastosowano się do wzorca MVVM z drobnymi różnicami względem nazewnictwa modułów. Jako model przyjęto tutaj obiekty DTO (ang. *Data Transfer Object*), które są tworzone podczas wymiany danych z serwerem za pomocą tzw. serwisów, czyli kontrolerów komunikacji z backendem. Więcej szczegółów omówiono w podrozdziale 5.5.2. Pozostałe moduły starano się odwzorować zgodnie ze wzorcem. Zastosowanie MVVM pozwoliło w łatwy sposób osiągnąć integralność z serwerem aplikacji przez co pracę na tym przebiegały znacznie szybciej.

5.5.2 Implementacja

Sposób w jaki została zaimplementowana aplikacja mobilna w pewnych aspektach przypomina odzwierciedlenie zasady działania aplikacji serwerowej. Jest to spowodowane pewnymi cechami wspólnymi używanej technologii, ale też faktem, że większość zespołu aplikacji mobilnej pracuje również przy implementacji backendu przez co nakładają się na siebie pewne wzorce. Istotna okazał się również fakt, że używana podczas implementacji technologia jest stosunkowo nowa, co ogranicza ilość dostępnych poradników i wskazówek. Mimo to implementacja aplikacji mobilnej zakończyła się sukcesem

5.5.2.1 Połączenie sieciowe

W omawianym systemie biblioteka Ktor odpowiedzialna jest za połączenie sieciowe. Umożliwia ona obsługę połączeń sieciowych w aplikacjach na system Android poprzez tworzenie instancji klienta

`HttpClient()`, który może konfigurować parametry połączenia (takie jak *timeout*, logowanie, obsługa JSON). Listing 56 prezentuje sposób w jaki tworzona jest taka instancja.

Listing 56 Tworzenie instancji HttpClient. Źródło: Opracowanie własne

```
private val client = HttpClient(CIO){
    install(ContentNegotiation) {
        json()
    }
    install(Logging) {
        logger = Logger.SIMPLE
        level = LogLevel.ALL
    }
    install(Resources)


}
```

Ktor automatycznie przeprowadza deserializację JSON, jeśli używany jest odpowiedni serializer. W tym przypadku za sprawą "*ContentNegotiation*" ustawiono domyślny serializer języka Kotlin. Odpowiedzi można mapować bezpośrednio na obiekty Kotlin.

Do wysyłania zapytań, takich jak GET, POST, PUT czy DELETE, wystarczy wywołać odpowiednią metodę o tej samej nazwie zawartą w obiekcie klienta. Listing 57 prezentuje w jaki sposób zaimplementowano wykonanie zapytania GET. Zastosowano tutaj również obsługę błędów które mogą powstać przy próbie połączenie z serwerem przez co na wczesnym etapie aplikacja jest w stanie wykryć niepożądane zachowanie.

Listing 57 Przykład wywołania zapytania GET. Źródło: Opracowanie własne

```
suspend fun get(path: String): Result<HttpResponse?> {
    return responseWrapper(
        try {
            client.get(path)
        } catch (e: Exception){
            println("[GET ERROR]: "+e.message)
            null
        }
    )
}
```

5.5.2.2 Bezpieczeństwo typów oraz połączenie z backendem

W celu nawiązania bezpośredniego połączenia z serwerem backendowym systemu zastosowano dodatkowe ustawienia klienta. Listing 58 prezentuje instalację dodatkowych pluginów, które urozmaicają funkcje obiektu. Przedstawiono tutaj obsługę domyślnego adresu oraz rodzaj oczekiwanych danych. Dodatkowo uwzględniono tutaj również automatyczną autoryzację w połączeniu z backendem za pomocą funkcji `getBearerTokens()`. Funkcja ta, jeżeli to możliwe zwraca token przechowywany w pamięci urządzenia. Token uwierzytelniający może zostać zapisany na urządzenie wyłącznie w momencie udanego logowania.

Listing 58 Domyślne ustawienia klienta Ktor. Źródło: Opracowanie własne

```
private val backendIp= "localhost"
private val backendPort= 12038
val backendUrl= "http://$backendIp:$backendPort"
```

```
    [...]


    defaultRequest {
        url(backendUrl)
        contentType(ContentType.Application.Json)
        accept(ContentType.Application.Json)
    }
    install(Auth) {


        bearer {
            loadTokens {
                getBearerTokens()
            }


            refreshTokens {
                getBearerTokens()
            }
        }
    }
```

Adresy końcówek, które używane są do komunikacji z serwerem backendowym całego systemu zdefiniowane zostały nie za pomocą zwykłych ciągów znaków (typu *string*) lecz poprzez definicję odpowiednich obiektów oraz adnotacji (Listing 59).

Listing 59 Przykład obiektu adresu końcówki. Źródło: Opracowanie własne

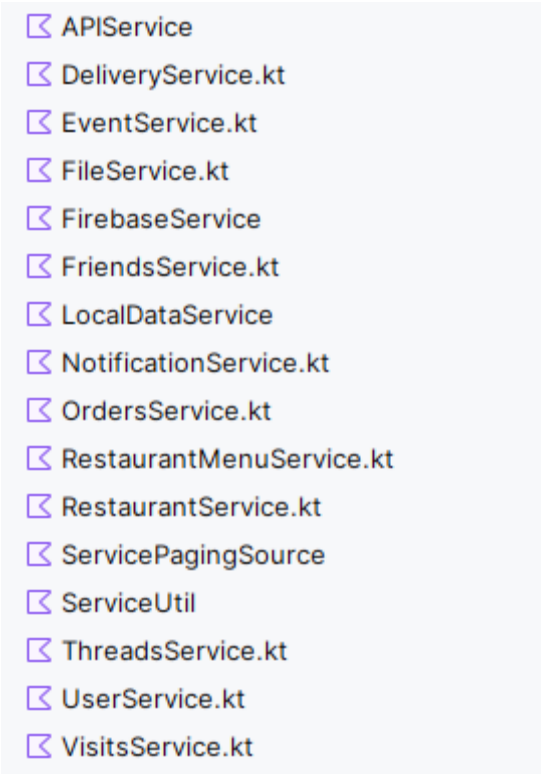
```
    @Resource("/friends")
    class Friends(val page: Int? = null, val perPage: Int? = null) {
        @Resource("{userId}")
        class UserId(val parent: Friends = Friends(), val userId: String) {
            @Resource("send-request")
            class SendRequest(val parent: UserId)
        }


        @Resource("{senderId}")
        class SenderId(val parent: Friends = Friends(), val senderId: String) {
            @Resource("mark-read")
            class MarkRead(val parent: SenderId)


            @Resource("accept-request")
            class AcceptRequest(val parent: SenderId)
        }


        @Resource("incoming")
        class Incoming(
            val parent: Friends = Friends(),
            val unreadOnly: Boolean? = null,
            val page: Int? = null,
            val perPage: Int? = null)


        @Resource("outgoing")
        class Outgoing(val parent:  Friends = Friends(), val page: Int? = null, val perPage:
Int? = null)
    }
```

Ktor potrafi skonstruować potrzebny adres końcówki na podstawie atrybutów oraz obiektów dziedziczących. W ten sposób każde dziedziczenie symbolizuje następny zasób końcówki. Pozwala to na osiągnięcie bezpieczeństwa typów, ponieważ większość nieprawidłowości może zostać rozwiązywane jeszcze na poziomie kompilacji kotlina. Następnie tak skonstruowane adresy mogą zostać przekazane jako atrybut do funkcji wykonującej zapytanie. Listing 60 ukazuje przykład metod odpowiedzialnych za odpowiednie wykonywanie zapytań GET, POST oraz PUT.

Listing 60 Przykład metod wykonujących zapytanie. Źródło: Opracowanie własne

```
    suspend inline fun <reified T : Any> get(resource: T): Result<HttpResponse?> {
        return responseWrapper(
            try {
                getHttpClient().get(resource)
            } catch (e: Exception){
                println("[GET ERROR]: "+e.message)
                null
            }
        )
    }
    suspend inline fun <reified T : Any> post(resource: T, obj: @Serializable Any):
Result<HttpResponse?> {
        return responseWrapper(
            try {
                getHttpClient().post(resource) {
                    setBody(obj)
                }
            } catch (e: Exception){
                println("[POST ERROR]: "+e.message)
                null
            }
        )
    }


    suspend inline fun <reified T : Any> put(resource: T, obj: @Serializable Any):
Result<HttpResponse?> {
        return responseWrapper(
            try {
                getHttpClient().put(resource) {
                setBody(obj)
            }
            } catch (e: Exception){
                println("[PUT ERROR]: "+e.message)
                null
            }
        )
    }
```

W celu pełnego wykorzystania przedstawionych mechanizmów łączenia z serwerem powstały również tzw. “serwisy”. Są to klasy stanowiące swego rodzaju interfejs między programistą, a serwerem umożliwiające wywoływanie końcówek w łatwy sposób. Serwisy zostały podzielone na kategorie zgodnie z ich przeznaczeniem. Rysunek 57 pokazuje sposób podziału serwisów.



Rysunek 57 Lista zaimplementowanych serwisów. Źródło: Opracowanie własne

Niektóre z serwisów dodatkowo działają niezależnie od serwera backendowego. Przykładem takich klas są:

- `FirebaseService` - klasa odpowiedzialna za uzyskanie odpowiednich tokenów potrzebych do integracji z usługami firebase,
- `LocalDataService` - pełni rolę kontrolera zapisu i odczytu danych z pamięci urządzenia.

Warto tutaj również zwrócić uwagę na `ServiceUtil`, jest to klasa abstrakcyjna, która zawiera podstawowe narzędzia opakowujące powtarzalne czynności walidacyjne. Zawiera ona również w sobie klasę odpowiedzialna za połączenie sieciowe (`APIService`) tak aby serwisy dziedziczące nie musiały jej za każdym razem inicjować.

Każdy z serwisów implementuje swój interfejs, aby programista w łatwy sposób mógł przejrzeć jakie funkcje są dostępne w ramach danej klasy. Jednym z dobrych przykładów serwisu jest `OrdersService`, który prezentuje Listing 61.

Listing 61 Implementacja OrdersService. Źródło: Opracowanie własne

```
interface IOrdersService{
   suspend fun getOrder(orderId: Any): Result<OrderDTO?>
   suspend fun cancelOrder(orderId: Any): Result<Boolean>
   suspend fun createOrder(order: OrderDTO): Result<OrderDTO?>
   suspend fun changeOrderStatus(orderId: Any, order: OrderDTO): Result<OrderDTO?>
}


class OrdersService():ServiceUtil(), IOrdersService {
   override suspend fun getOrder(orderId: Any): Result<OrderDTO?> {
       val res = api.get(Orders.OrderId(orderId=orderId.toString()))
       return complexResultWrapper(res)
   }
```

```
    override suspend fun cancelOrder(orderId: Any): Result<Boolean> {
        val res = api.post(Orders.OrderId.Cancel(parent = 
Orders.OrderId(orderId=orderId.toString())), "")
        return booleanResultWrapper(res)
    }


    override suspend fun createOrder(order: OrderDTO): Result<OrderDTO?> {
        val res = api.post(Orders(), order)
        return complexResultWrapper(res)
    }


    override suspend fun changeOrderStatus(orderId: Any, order: OrderDTO): 
Result<OrderDTO?> {
        val res = api.put(Orders.OrderId.Status(
            parent = Orders.OrderId(orderId=orderId.toString())
        ), order)
        return complexResultWrapper(res)
    }
}
```

5.5.2.3 Nawigacja po aktywnościach

Po uruchomieniu aplikacji mobilnej użytkownikowi ukazuje się ekran ładowania, na którym decydowane jest, czy użytkownik był już wcześniej zalogowany oraz czy jego token JWT nie utracił swojej ważności. W zależności od tych czynników przekierowywany jest do ekranu logowania lub głównego panelu aplikacji. Listing 62 prezentuje przebieg omawianego mechanizmu.

Listing 62 Punkt startowy aplikacji. Źródło: Opracowanie własne

```
override fun onCreate(savedInstanceState: Bundle?) {
    val splashScreen = installSplashScreen()
    super.onCreate(savedInstanceState)
    var isLoading = true
    var startPoint: @Composable ()->Unit
    splashScreen.setKeepOnScreenCondition { isLoading }

    lifecycleScope.launch {
        askNotificationPermission()

        startPoint = if(LoginViewModel().refreshToken()) {
             if(Roles.RESTAURANT_EMPLOYEE in UserService.UserObject.roles) {
                 { EmployeeHomeActivity() }
             }
             else{
                 { HomeActivity() }
             }
         } else {
             { LandingActivity() }
         }

        setContent {
            AppTheme {
                isLoading=false
                startPoint()
            }
        }
    }
}
```

Warto tutaj zaznaczyć, że jeżeli użytkownik jest zalogowany, sprawdzana jest również jego rola w systemie (np. czy jest to zwykły użytkownik czy pracownik restauracji) i na bazie tego aplikacja wyświetla stosowny ekran.

Biblioteka compose przewiduje nawigację między ekranami przy użyciu komponentu `NavHost`. Obiekt ten pozwala zdefiniować poszczególne ścieżki oraz kontroler, który przekazywany jest odpowiednim ekranom jako atrybut (Listing 63).

Listing 63 Przykład oobiektu NavHost. Źródło: Opracowanie własne

```
    NavHost(navController = innerNavController, startDestination = MainRoutes.Home,
modifier = Modifier.padding(it)){
        composable<MainRoutes.Home>{
            MapActivity()
        }
        composable<RestaurantManagementRoutes.Restaurant>{
            RestaurantManagementActivity(navControllerHome = innerNavController)
        }
        composable<RegisterRestaurantRoutes.Register>{
            RegisterRestaurantActivity(navControllerHome = innerNavController)
        }
        composable<MainRoutes.Settings>{
            SettingsActivity(homeNavController = innerNavController, themeChange =
{ darkTheme = !darkTheme } )
        }
        composable<RestaurantRoutes.Reservation>{
            RestaurantReservationActivity(navController = innerNavController)
        }
        composable<MainRoutes.ChatList> {
            ChatListActivity()
        }
        composable<AuthRoutes.Landing>{
            LaunchedEffect(Unit) {
                bottomBarState.value = false
            }
            LandingActivity()
        }
    }
```

Ścieżki prowadzące do poszczególnych ekranów, podobnie jak w obsłudze końcówek również zdefiniowane zostały pod względem bezpieczeństwa typów. Mechanizm ten działa bliźniaczo podobnie i jego przykład prezentuje Listing 64.

Listing 64 Przykład adresu NavHost. Źródło: Opracowanie własne

```
    data object UserRoutes {


        @Serializable
        object ChatList


        @Serializable
        data class Chat(
            val threadId: Int,
            val threadTitle: String
        )
```

```
    @Serializable
    data class UserProfile(
        val userId: String
    )


    @Serializable
    object Ticket


    @Serializable
    object TicketHistory


    @Serializable
    object Orders


    @Serializable
    object FindFriends
}
```

5.5.2.4 Przygotowanie danych

Zanim pobrane od serwera backendowego dane zostaną wyświetlone na ekranie użytkownika aplikacja zgodnie z modelem MVVM przygotowuje je za pomocą viewmodelu. Są to specjalne klasy, które służą do odpowiednio przygotowanych informacji. Na tym etapie przeprowadzana jest również wstępna walidacja danych przy pomocy utworzonej na potrzeby systemu klasy `Result` (Listing 65). Odpowiada on za określenie czy w trakcie pobierania danych wystąpił błąd i jeżeli tak, to przechowuje jego treść. Warto tutaj zaznaczyć, że błędów z jednego zapytania może być kilka, dlatego też klasa `Result` jest do tego przystosowana.

Listing 65 Implementacja klasy Result. Źródło: Opracowanie własne

```
class Result<T>(
    var isError: Boolean,
    val value: T
){
    constructor(
        isError: Boolean,
        errors: JSONObject? = null,
        value: T
    ): this(
        isError=isError,
        value=value
    ){
        this.errors = errors?.keys()?.asSequence()?.associateWith {
            Integer.parseInt(errors.getJSONArray(it)[0].toString())
        }
    }


    constructor(
        isError: Boolean,
        errors: Map<String, Int>? = null,
        value: T
    ) : this(
        isError=isError,
        value=value
    ){
        this.errors = errors
```

```
    }


    var errors: Map<String, Int>? = null
        get() = if (!isError) throw UnsupportedOperationException() else field


}
```

Klasy typu viewmodel oprócz wstępnej walidacji obsługują różne potrzebne pod kątem wyświetlania typy czynności:

- definiowanie zmiennych pomocniczych,
- zarządzanie stanem ładowania danych,
- asynchroniczne pozyskiwanie oraz wysyłanie danych,
- utrzymanie danych nawet podczas zmian orientacji ekranu,
- definicję logiki biznesowej,
- aktualizowanie danych w razie takiej potrzeby.

Jednym z dobrych przykładów implementacji klasy typu *viewmodel* jest Listing 66, który reprezentuje klasę `SocialViewModel`. Klasa ta w sprawny sposób przygotowuje dane pod paginację dostarczając obiekt `StateFlow` za pośrednictwem klasy `Result`. Jest to również jeden z prostszych przykładów mający na celu zaprezentować mechanikę działania przygotowania danych.

Listing 66 Przykład implementacji viewmodelu. Źródło: Opracowanie własne

```
class SocialViewModel(
    private val userService: IUserService = UserService()
): ReservantViewModel() {


    private val _users = MutableStateFlow<PagingData<FoundUserDTO>>(PagingData.empty())
    val users: StateFlow<PagingData<FoundUserDTO>> = _users.asStateFlow()
    private val _threads = MutableStateFlow<PagingData<ThreadDTO>>(PagingData.empty())
    var threads: StateFlow<PagingData<ThreadDTO>> = _threads.asStateFlow()


    var threadQuery = mutableStateOf("")
    var userQuery = mutableStateOf("")


    init {
        viewModelScope.launch {
            getThreads(threadQuery.value)
            getUsers(userQuery.value)
        }
    }


    suspend fun getPhoto(url: String): Bitmap?{
        val result = fileService.getImage(url)
        if (!result.isError){
            return result.value!!
        }
        return null
    }
```

```
    suspend fun getUsers(query: String){
        val res = userService.getUsers(name = query)


        if (res.isError || res.value == null){
            throw Exception()
        }


        res.value.cachedIn(viewModelScope).collect{
            _users.value = it
        }
    }


    suspend fun getThreads(query: String) {
        val res = userService.getUserThreads()


        if (res.isError || res.value == null){
            throw Exception()
        }


        res.value.cachedIn(viewModelScope).collect {
            _threads.value = it.filter { thread ->
                thread.title?.contains(query) ?: false ||
                thread.participants?.any { participant ->
                    participant.firstName.contains(query)
                } ?: false
            }
        }
    }
}
```

Można tutaj zaobserwować, że obiekt ten dziedziczy z innej klasy, `ReservantViewModel`, która dostarcza potrzebne narzędzia często używane przez pozostałe viewmodele używane w omawianej aplikacji tj. obsługa pobierania zdjęć lub walidowanie elementów formularza. Jest to implementacja dziedzicząca z abstrakcyjnej klasy `ViewModel`, która stanowi podstawowe narzędzie dostarczone przez SDK Androida. Listing 67 prezentuje przykład jej implementacji.

Listing 67 Przykład nadrzędnego viewmodelu. Źródło: Opracowanie własne

```
open class ReservantViewModel(
    val fileService: FileService = FileService()
): ViewModel() {


    suspend fun fetchPhoto(photo: String): Bitmap? {
        val result = fileService.getImage(photo)


        return if (!result.isError) {
            result.value
        } else {
            null
        }
    }


    protected fun <T> getFieldError(result: Result<T>, name: String): Int {
```

```
            if (!result.isError) {
                return -1
            }


            return result.errors?.getOrDefault(name, -1) ?: -1
        }


        fun<T> getToastError(result: Result<T>): Int {
            return getFieldError(result, "TOAST")
        }


        protected fun isInvalidWithRegex(regex: String, str: String): Boolean {
            return !Pattern.matches(regex, str)
        }
    }
```

5.5.2.5 Komponowanie interfejsu użytkownika

Biblioteka compose umożliwia swojego rodzaju programowanie interfejsu za pomocą tzw. komponentów. Są to specjalne funkcje, które dzięki zdefiniowanymi w nich atrybutom pozwalają programistom na modyfikację poszczególnych aspektów danego obiektu. Atrybutów może być wiele i mogą one posiadać różne typy, jednak szczególnym przypadkiem jest tzw. `modifier`, atrybut pozwalający na zmianę ogólnej formy, położenia lub zachowań komponentu w określonych warunkach.

Listing 68 Ekran dodawania recenzji. Źródło: Opracowanie własne

```
    val reviewsViewModel = viewModel<ReviewsViewModel>(
        factory = object : ViewModelProvider.Factory {
            override fun <T : ViewModel> create(modelClass: Class<T>): T =
                ReviewsViewModel(restaurantId) as T
        }
    )


    var stars by remember { mutableStateOf(0) }
    var contents by remember { mutableStateOf("") }
    var isError by remember { mutableStateOf(false) }


    Column(
        modifier = Modifier
            .padding(16.dp)
    ) {
        IconWithHeader(
            icon = Icons.Rounded.Add,
            text = stringResource(id = R.string.label_add_review),
            showBackButton = true,
            onReturnClick = { navController.navigate(RestaurantRoutes.Details(restaurantId =
restaurantId)) }
        )
        Text(
            text = stringResource(id = R.string.label_rating),
            style = MaterialTheme.typography.bodyLarge.copy(
                fontWeight = FontWeight.Bold
            ),
            modifier = Modifier.padding(8.dp)
        )
        Spacer(modifier = Modifier.height(8.dp))
        Row {
```

```
            for (i in 1..5) {
                Icon(
                    imageVector = if (i <= stars) Icons.Filled.Star else
Icons.Filled.StarBorder,
                    contentDescription = "$i Star",
                    tint = if (i <= stars) MaterialTheme.colorScheme.secondary else
MaterialTheme.colorScheme.primary,
                    modifier = Modifier
                        .size(40.dp)
                        .clickable { stars = i }
                )
            }
        }
        Spacer(modifier = Modifier.height(16.dp))


        FormInput(
            inputText = contents,
            onValueChange = { contents = it },
            label = stringResource(id = R.string.label_review_content),
            modifier = Modifier.fillMaxWidth(),
            isError = isError,
            errorText = stringResource(id = R.string.error_duplicate_review)
        )


        Spacer(modifier = Modifier.height(16.dp))


        ButtonComponent(
            onClick = {
                if (contents.isNotBlank()) {
                    reviewsViewModel.viewModelScope.launch {
                        reviewsViewModel.addReview(stars, contents)
                        if (reviewsViewModel.result.isError) {
                            isError = true
                        } else {
                            isError = false
                            navController.popBackStack()
                        }
                    }
                }
            },
            label = stringResource(id = R.string.label_add_review),
            modifier = Modifier.fillMaxWidth()
        )
    }
```

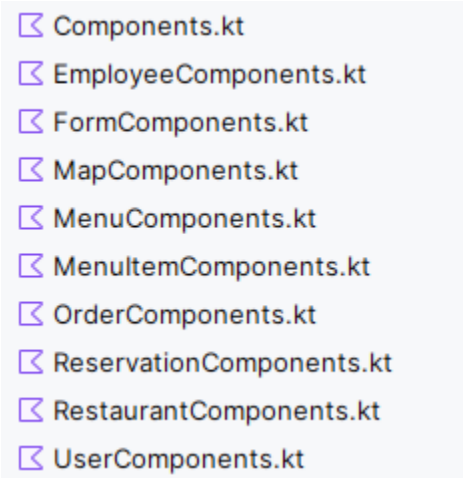
Listing 68 pokazuje w jaki sposób zaimplementowano ekran komponowany za pomocą wywoływania funkcji oraz przypisania odpowiednich komponentów. Oprócz specyficznych dla danych komponentów atrybutów warto zwrócić tutaj uwagę na sposób w jaki zdefiniowano atrybut `modifier`. Jest on obecny w każdym z komponentów i pełni uniwersalną rolę.

Biblioteka w bardzo łatwy sposób umożliwia definiowanie własnych komponentów. Zazwyczaj sprowadza się to opakowywania gotowych już podstawowych komponentów w specyficzny dla danego zastosowania sposób. W celu stworzenia takiego komponentu wystarczy zdefiniować zwykłą funkcję oraz dodać do niej adnotację `@Composable`, tak jak pokazuje to Listing 69. Przy definiowaniu własnego komponentu decyzja czy powinien się znajdować tam atrybut `modifier` należy do nas, warto jednak dodawać go, jeżeli jest to możliwe.

Listing 69 Przykład niestandardowego komponentu. Źródło: Opracowanie własne

```
@Composable
fun RestaurantCard(
   onClick: () -> Unit,
   name: String,
   location: String,
   city: String,
   image: ImageBitmap?) {
   Card(
       modifier = Modifier
           .padding(16.dp)
           .fillMaxWidth()
           .wrapContentHeight()
           .clickable { onClick() },
       elevation = CardDefaults.cardElevation(
           defaultElevation = 6.dp
       ),
       shape = RoundedCornerShape(16.dp)) {
       Row(
           modifier = Modifier
               .background(MaterialTheme.colorScheme.surface)
               .fillMaxWidth()
       ) {
           if(image != null){
               Image(
                   bitmap = image,
                   contentDescription = null,
                   modifier = Modifier
                       .height(100.dp)
                       .width(100.dp),
                   contentScale = ContentScale.Crop,
                   alignment = Alignment.Center
               )
           }
           else {
               Image(
                   painter = painterResource(R.drawable.restaurant_template_icon),
                   contentDescription = null,
                   modifier = Modifier
                       .height(100.dp)
                       .width(100.dp),
                   contentScale = ContentScale.Crop,
                   alignment = Alignment.Center
               )
           }
           Column(
               modifier = Modifier
                   .padding(16.dp)
           ) {
               Text(
                   text = name,
                   style = MaterialTheme.typography.headlineMedium,
                   fontWeight = FontWeight.Bold
               )
               Text(
                   text = "$location, $city",
                   style = MaterialTheme.typography.bodyMedium,
                   color = Color.Gray
               )
           }
       }
   }
}
```

W omawianym systemie tworzenie własnych komponentów jest dosyć powszechną praktyką dlatego ze względu na ich ilość zdecydowano się podzielić je pod względem tematyki zastosowania. Rysunek 58 ilustruje zestawy przygotowanych komponentów.



Rysunek 58 Podział komponentów. Źródło: Opracowanie własne

6 Przykłady użycia i funkcjonalności

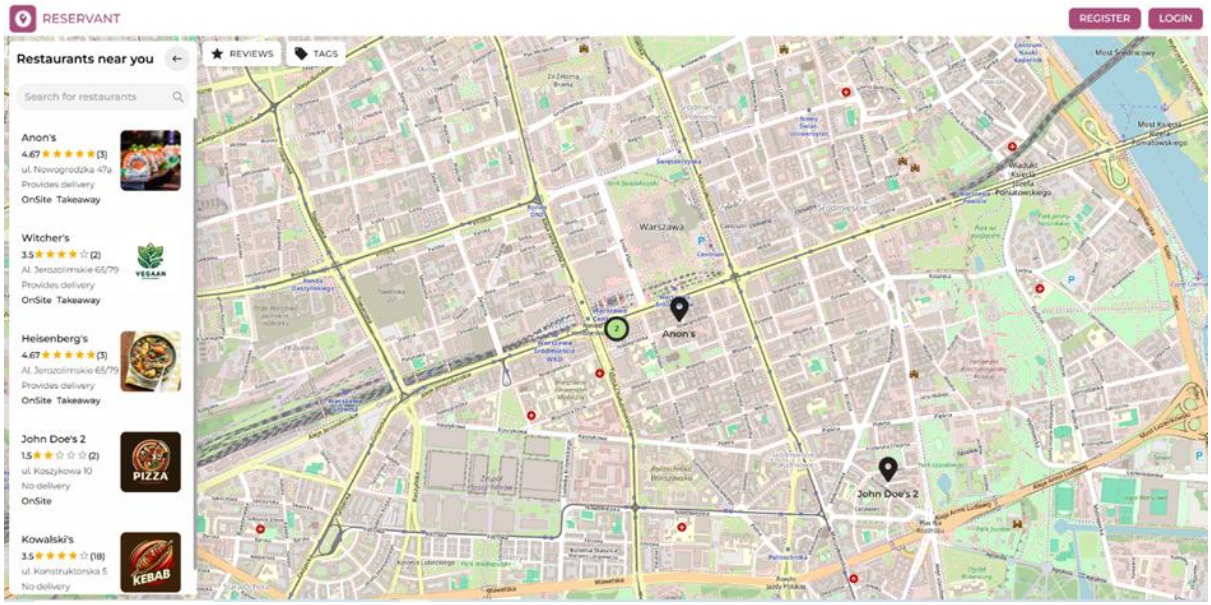
Rozdział ten prezentuje praktyczne zastosowania interfejsu użytkownika stworzonego systemu. Omawia kluczowe funkcje dostępne dla użytkownika, takie jak nawigacja, obsługa danych wejściowych i wizualizacja wyników. Zawarte zostały tutaj przykłady ilustrujące typowe scenariusze użytkowania.

6.1 Aplikacja webowa

Aplikacja webowa systemu Reservant to interaktywna platforma internetowa wspierająca restauracje w procesie rezerwacji stolików, składania zamówień czy kontrolowania stanu magazynu bezpośrednio z poziomu aplikacji. Funkcjonalności aplikacji różnią się w zależności od aktorów.

6.1.1 Ekran startowy gościa

Ekran startowy, który przedstawia Rysunek 59 prezentowany jest dla niezalogowanego użytkownika, pełni on rolę wprowadzenia do aplikacji, umożliwiając przeglądanie restauracji oraz zapoznanie się z jej podstawowymi funkcjami. Aplikacja udostępni użytkownikowi całość funkcjonalności dopiero po zalogowaniu.

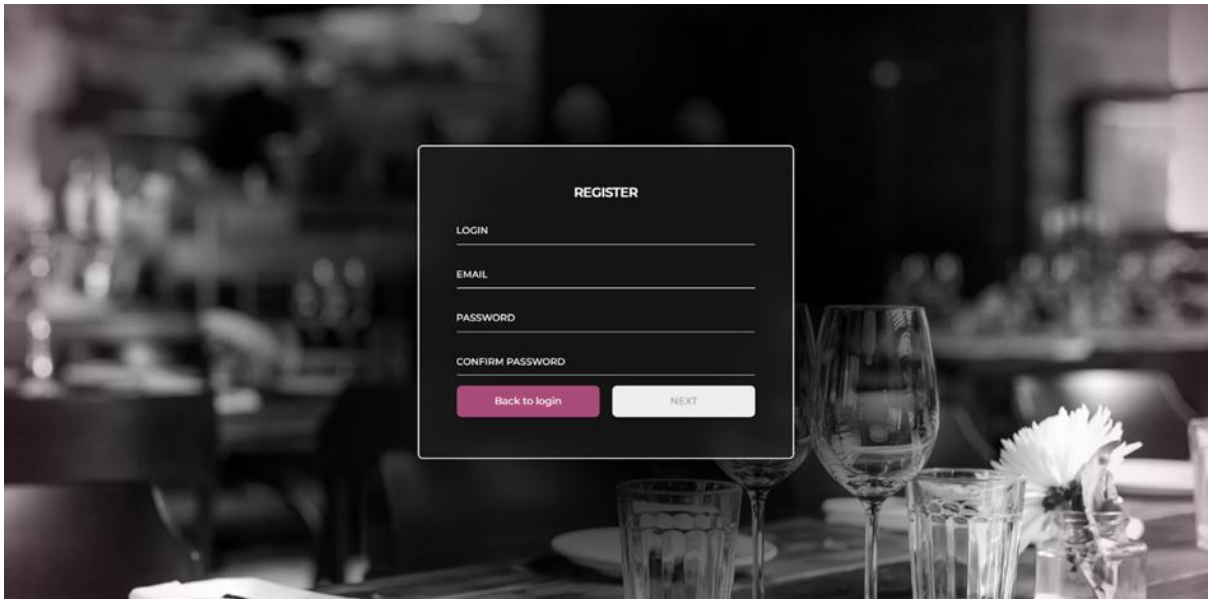


Rysunek 59 Ekran startowy aplikacji webowej. Źródło: Opracowanie własne

Niezalogowany gość oprócz możliwości sprawdzenia informacje o dostępnych placówkach gastronomicznych, może zapoznać się z menu poszczególnych restauracji lub sprawdzić zdjęcia udostępnione przez restauracje.

6.1.2 Logowanie/Rejestracja

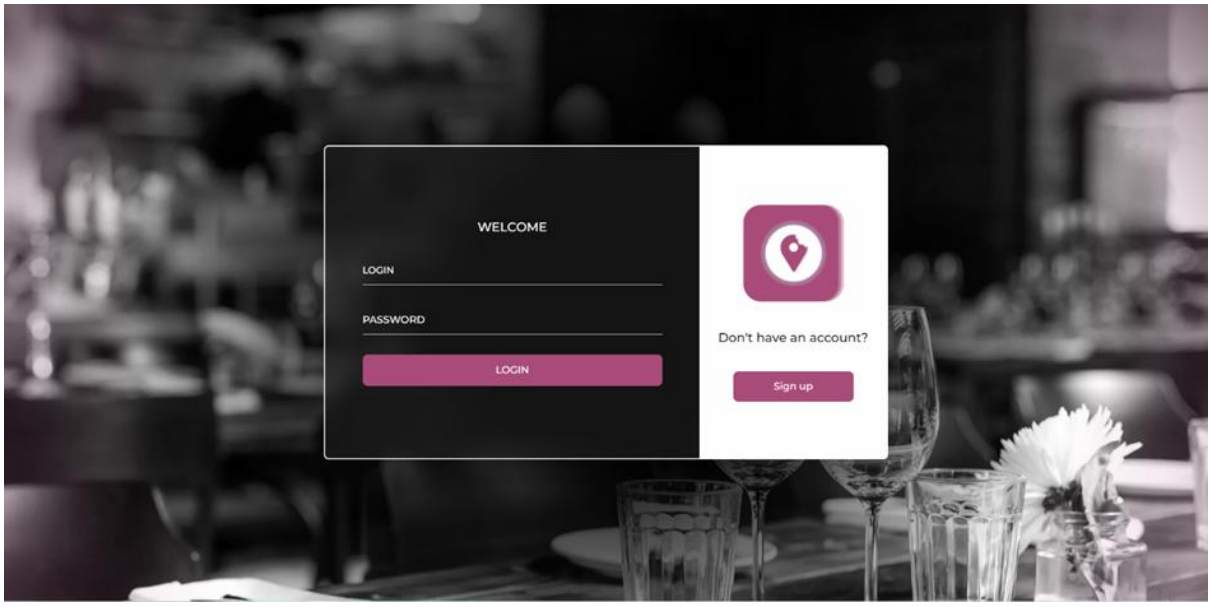
Pierwszym krokiem, aby odblokować szereg dodatkowych funkcjonalności zawartych w aplikacji Reservant jest zalogowanie się po ówczesnej rejestracji. Klienci goszczący w aplikacji Reservant po raz pierwszy powinni przejść wstępną rejestrację. Ekran do tego służący zaprezentowany został przez Rysunek 60.



Rysunek 60 Ekran rejestracji nowego użytkownika. Źródło: Opracowanie własne

Ekran ten wymaga od klienta wprowadzenie kilku kluczowych danych:

- Login – Jest to uniwersalna nazwa konta wykorzystywana między innymi do logowania,
- E-Mail - Użytkownik musi podać prawidłowy adres w formacie zgodnym powszechnymi normami. Adres e-mail pełni funkcję unikalnego identyfikatora konta oraz może być wykorzystywany w przyszłości między innymi do resetowania hasła. W przypadku wpisania nieprawidłowego adresu e-mail system wyświetla komunikat błędu: „*Invalid e-mail*”,
- Hasło – Musi spełniać wymagania bezpieczeństwa, które zostały określone przez system takie jak;
    - Minimum 8 znaków,
    - Co najmniej jedna wielka litera,
    - Co najmniej jedna mała litera,
    - Co najmniej jedna cyfra,
    - Co najmniej jeden znak specjalny,
- W przypadku niespełnienia tych kryteriów, system wyświetla komunikat błędu: „*Password must contain 8 characters, one uppercase, one lowercase, one number and one special character*”,
- Potwierdzenie hasła – Użytkownik musi ponownie wpisać hasło, aby potwierdzić jego poprawność. Wprowadzone dane muszą być identyczne z hasłem podanym w poprzednim polu. W przypadku wprowadzenia innego hasła wyświetlony zostaje komunikat „*Passwords must match*”, natomiast w przypadku pozostawienia wspomnianego pola pustego wyświetla się komunikat „*Passwords confirmation is required*”.

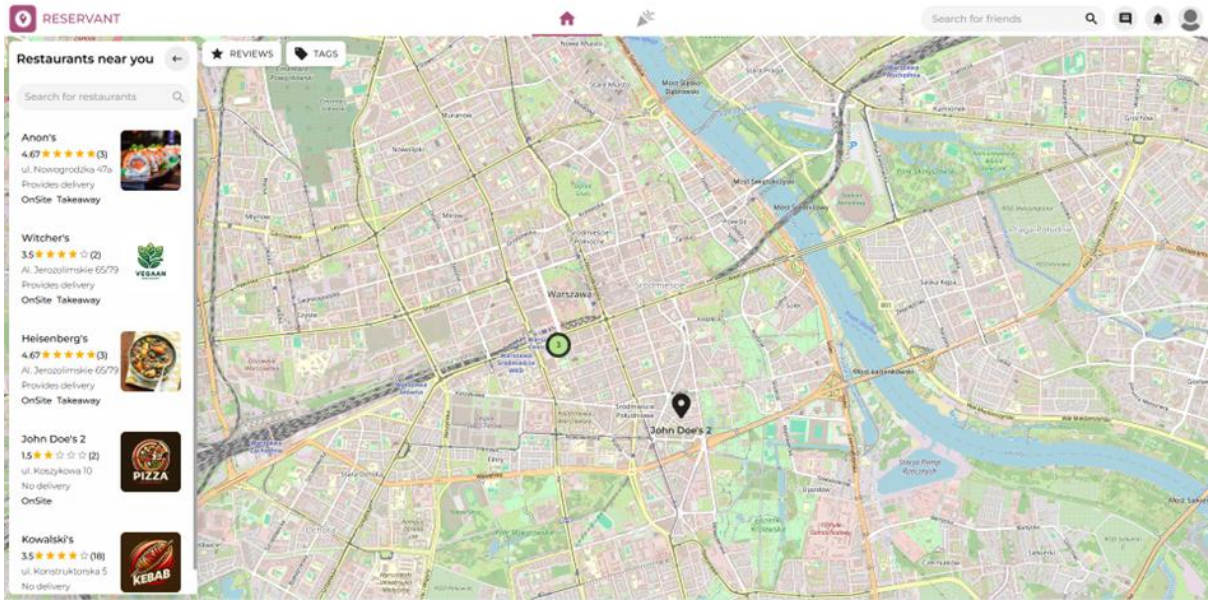


Rysunek 61 Ekran logowania użytkownika. Źródło: Opracowanie własne

Rysunek 61 prezentuje panel logowania użytkownika. Klient nieposiadający konta może skorzystać z opcji rejestracji znajdującej się pod przyciskiem „*Sign up*”. Użytkownicy posiadający konta wypełniają pola obowiązkowe „*Login*” (Unikalny identyfikator klienta) oraz „*Password*” (Hasło ustalone w fazie rejestracji) następnie zatwierdzają przyciskiem Login.

6.1.3 Widok zalogowanego użytkownika

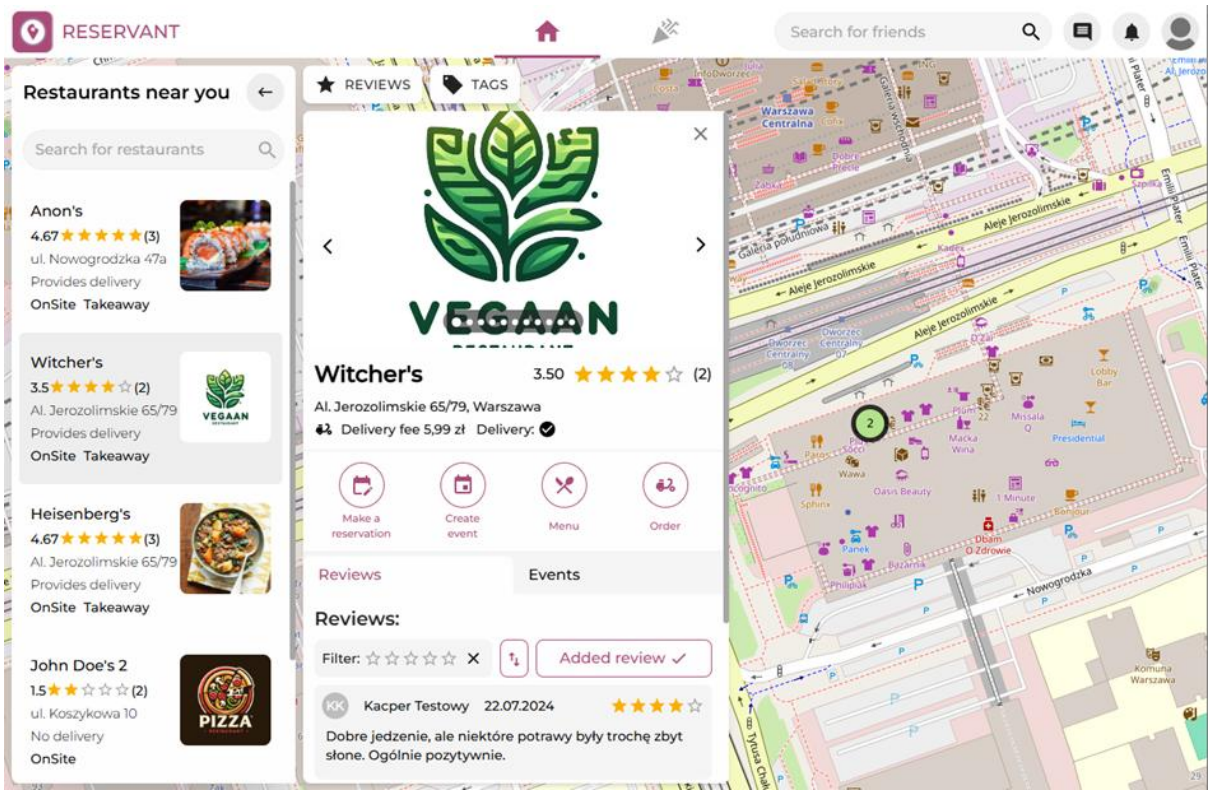
Po pomyślnym zalogowaniu się wyświetlona zostaje strona startowa zalogowanego użytkownika zawierająca mapę wraz z zaznaczonymi na niej restauracjami, co przedstawia Rysunek 62.



Rysunek 62 Widok zalogowanego użytkownika. Źródło: Opracowanie własne

Na interaktywnej mapie klienci mogą dowolnie przeglądać dostępne restauracje. Strona ta wyposażona jest w następujące elementy:

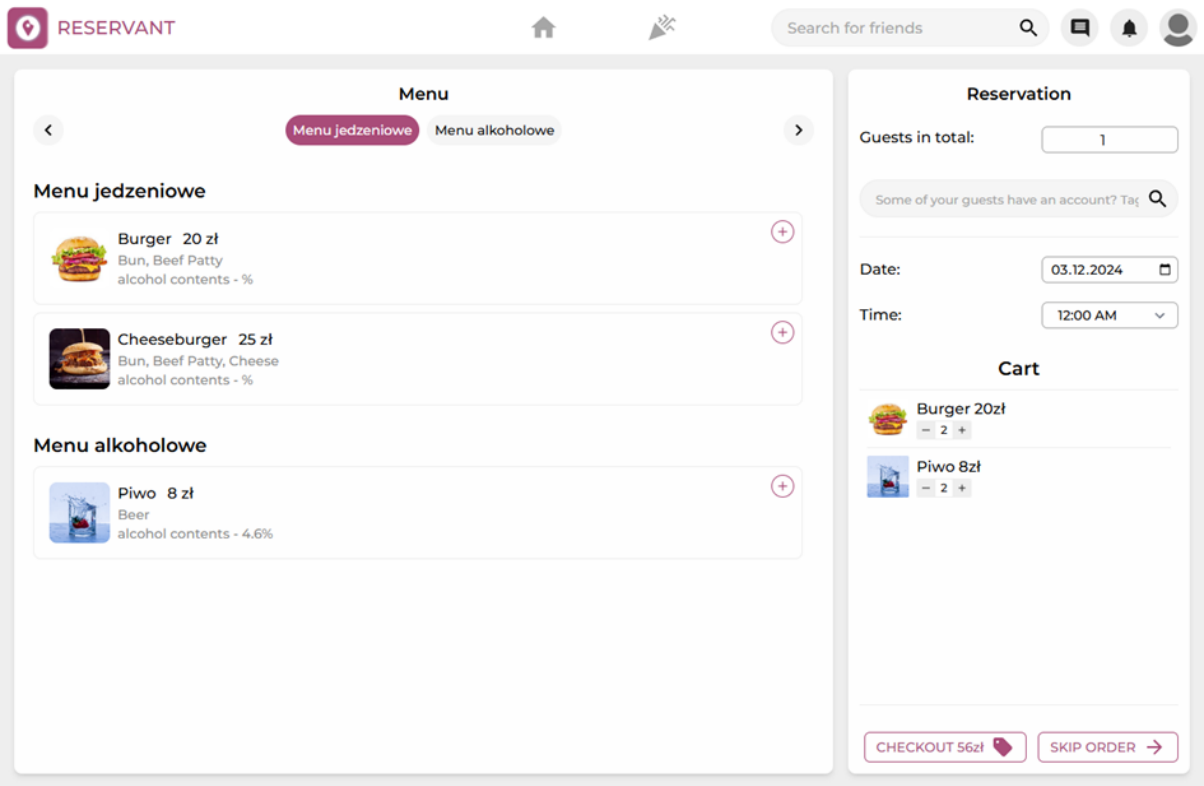
- Górny pasek nawigacyjny;
    - Z prawej strony znajduje się pole wyszukiwania "*Search for friends*", umożliwiające wyszukiwanie znajomych,
    - Obok pola wyszukiwania umieszczono ikonę profilu, wiadomości oraz powiadomień,
- Lewy panel boczny – Lista restauracji;
    - W lewej części ekranu znajduje się lista restauracji, zawierająca zdjęcia, nazwy restauracji, adresy, oceny oraz informacje o opcjach dostawy (np. "*OnSite*", "*Takeaway*", "*Delivery*"),
    - W górnej części tego panelu znajduje się pasek wyszukiwania "*Search for restaurants*", pozwalający użytkownikowi na przefiltrowanie listy restauracji,
- Centralny panel – Szczegóły wybranej restauracji (Rysunek 63);
    - Zdjęcia reprezentujące restaurację (w formie przewijanych zdjęć),
    - Nazwę restauracji, adres, ocenę w postaci gwiazdek oraz liczbę opinii,
    - Informacje o opłacie za dostawę i dostępnych opcjach dostawy (np. "*Delivery*"),
    - Ikony opcji takich jak „*Make a reservation*”, „*Create event*”, "*Menu*" oraz "*Order*",
    - Recenzje napisane przez użytkowników (zawierające datę oraz liczbę gwiazdek).



Rysunek 63 Centralny panel strony główne. Źródło: Opracowanie własne

6.1.4 Złożenie rezerwacji

Zalogowany klient po wejściu w centralny panel restauracji ma możliwość złożenia rezerwacji. Odbywa się to poprzez kliknięcie w przycisk „Zarezerwuj”. Rezerwacja może zawierać dodatkowe zamówienie z dostępnego menu. W celu finalizacji klient musi wybrać dostępną datę oraz godzinę rezerwacji. W przypadku dodatkowego zamówienia klient musi opłacić wstępny rachunek klikając w przycisk „*Checkout*”. Dostępne menu wraz z możliwością dokonania rezerwacji przedstawia Rysunek 64.

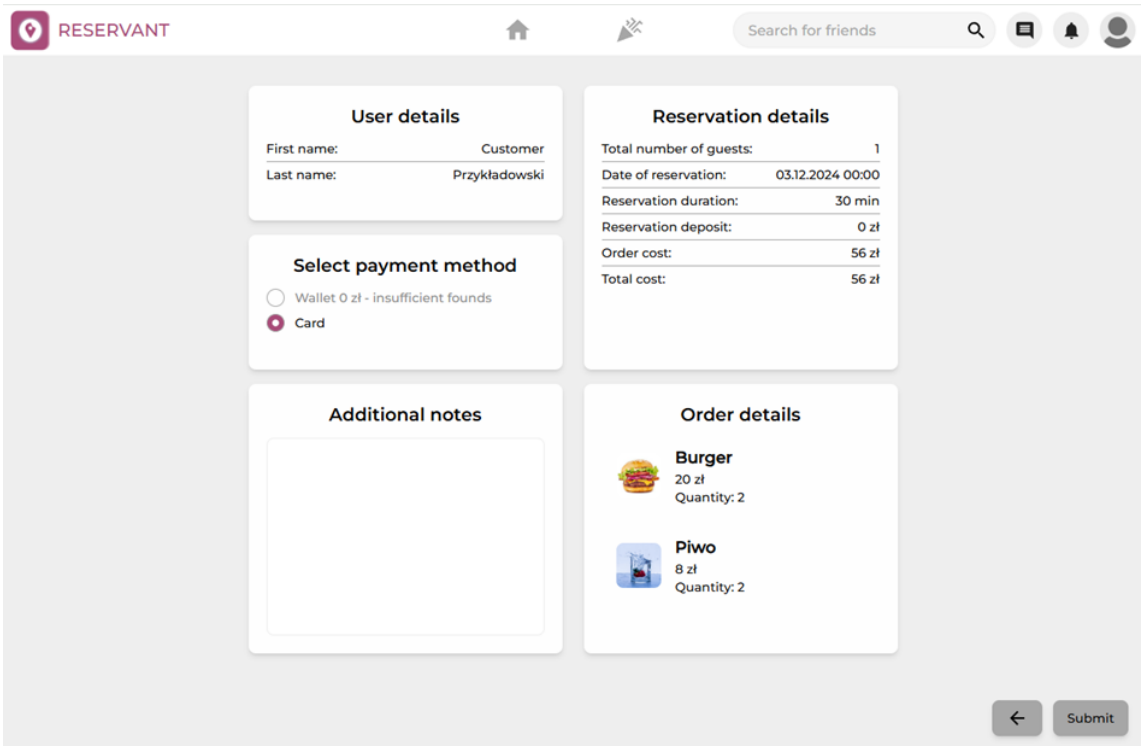


Rysunek 64 Ekran złożenia rezerwacji. Źródło: Opracowanie własne

Po kliknięciu w „*Checkout*” użytkownik przeniesiony zostaje do ekranu podsumowania rezerwacji (Rysunek 65). Na ekranie wyświetlone zostają najważniejsze informacje dotyczące zamówienia takie jak:

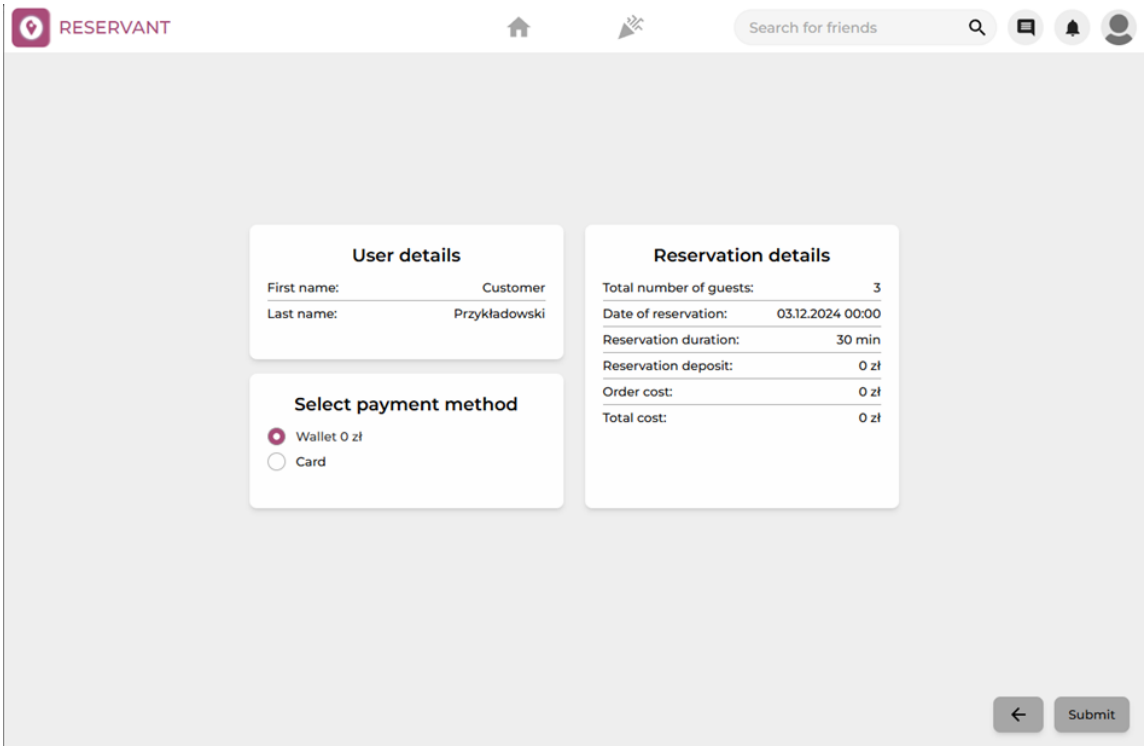
- Dane rezerwującego,
- Metoda płatności,
- Szczegóły rezerwacji,
- Notatka dla restauracji,
- Szczegóły zamówienia.

Następnie po kliknięciu w przycisk „*Submit*” rezerwacje z zamówieniem zostaje przekazana do realizacji.



Rysunek 65 Podsumowanie rezerwacji z zamówieniem. Źródło: Opracowanie własne

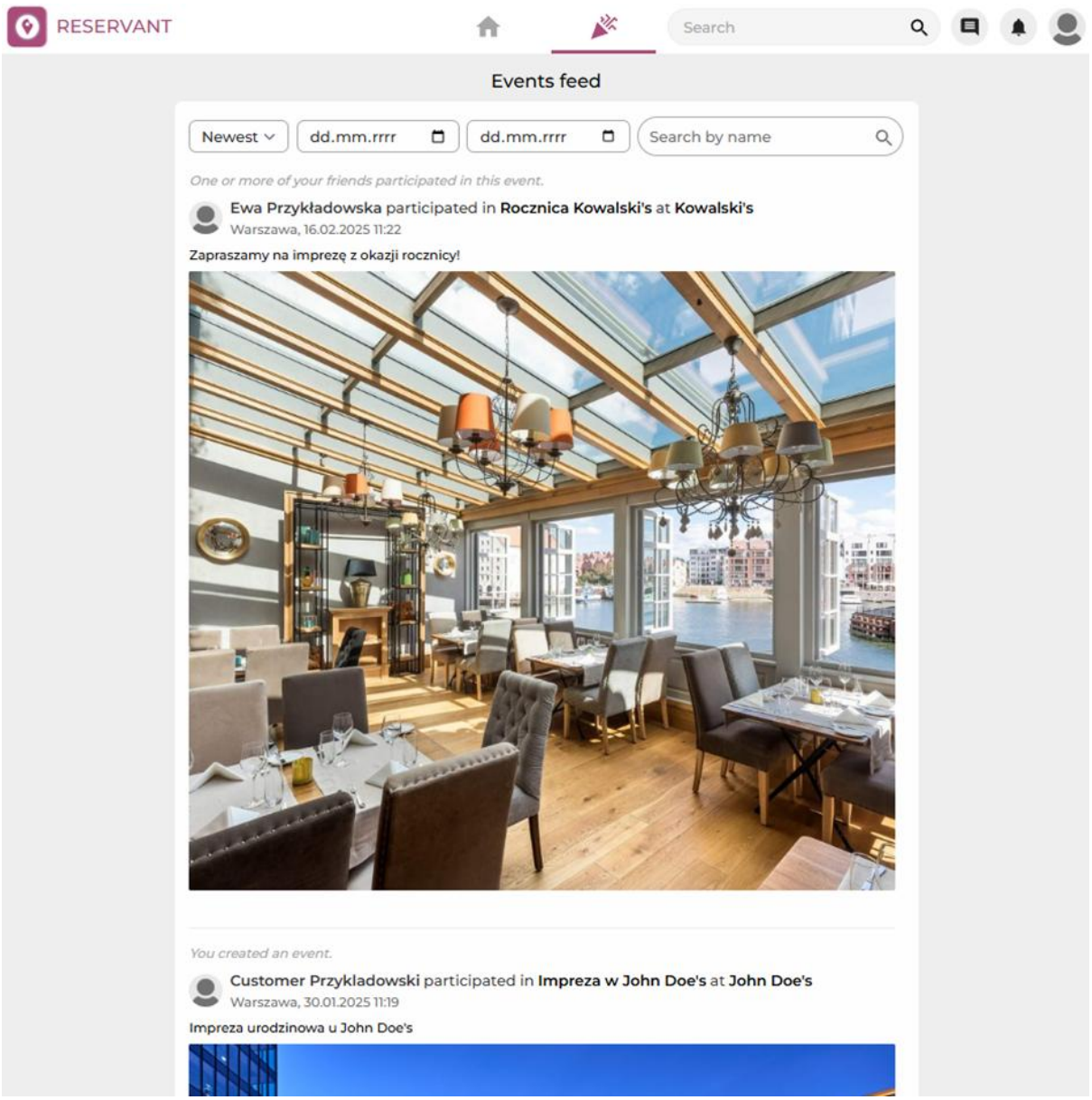
Klienci mają również możliwość złożenia rezerwacji bez zamówienia. Przy takim rozwiązaniu opłata nie jest naliczana, wystarczy podać jedynie datę, godzinę oraz liczbę osób. Na podsumowaniu znajdują się wtedy jedynie szczegóły użytkownika, płatność wynosząca 0 zł oraz szczegóły rezerwacji zgodnie z wybranymi preferencjami co ukazuje Rysunek 66.



Rysunek 66 Podsumowanie rezerwacji bez zamówienia. Źródło: Opracowanie własne

6.1.5 Wydarzenia

Sekcja ta prezentuje użytkownikowi dostępne wydarzenia, które odbywają się w zarejestrowanych w systemie lokalach (Rysunek 67). Są to imprezy lub spotkania otwarte dla społeczności aplikacji Reservant.

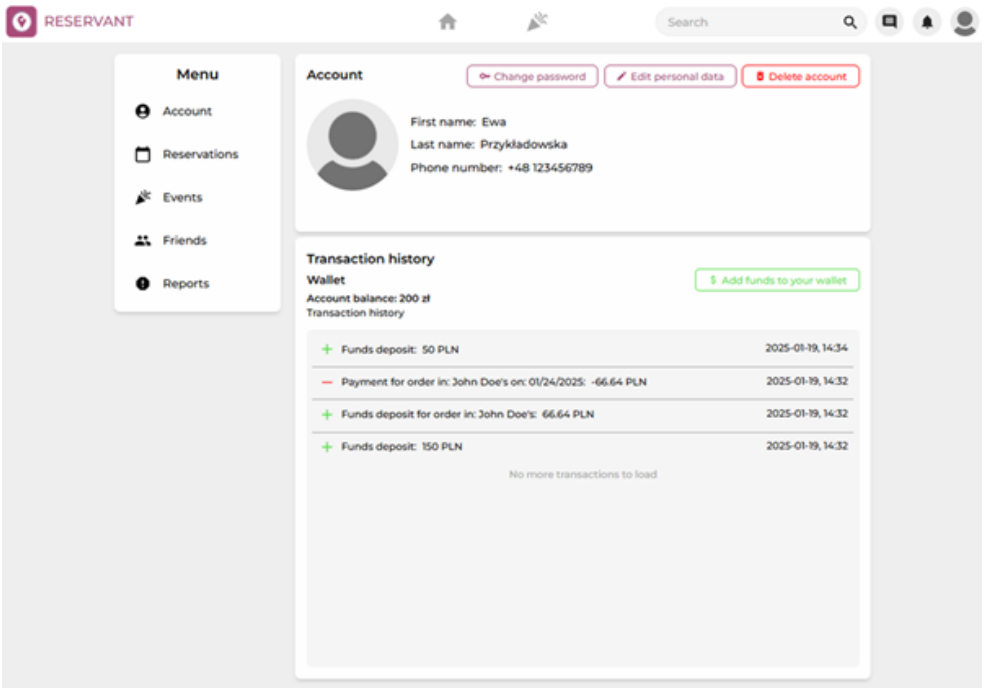


Rysunek 67 Sekcja wydarzenia. Źródło: Opracowanie własne

Klienci mogą filtrować wydarzenia wydarzenia, sortując je według kolejności ich dodawania, daty rozpoczęcia za pomocą pola tekstowego „*search by name*”.

6.1.6 Panel użytkownika

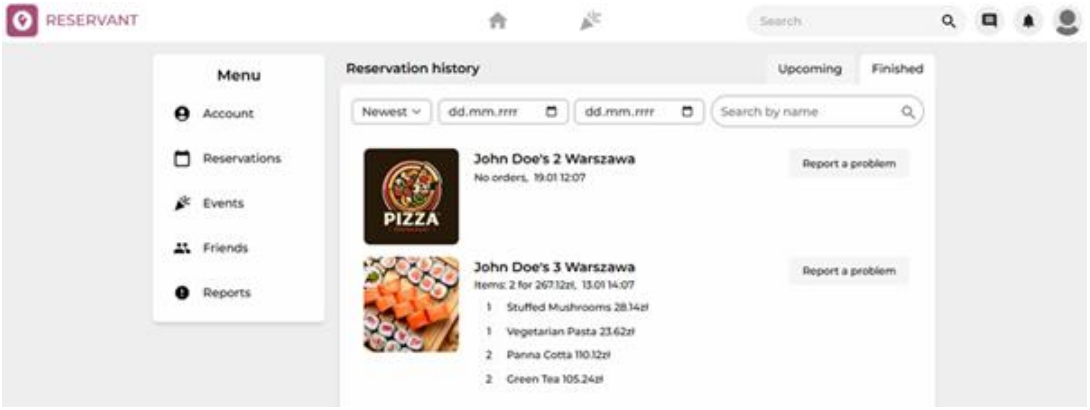
Panel ten przeznaczony jest do podsumowania danych użytkownika i jego transakcji, sprawdzenia zarówno przeszłych jak i przyszłych rezerwacji, listy wydarzeń, w których klient bierze lub brał udział oraz zarządzania listą znajomych.



Rysunek 68 Sekcja zarządzania kontem użytkownika. Źródło: Opracowanie własne

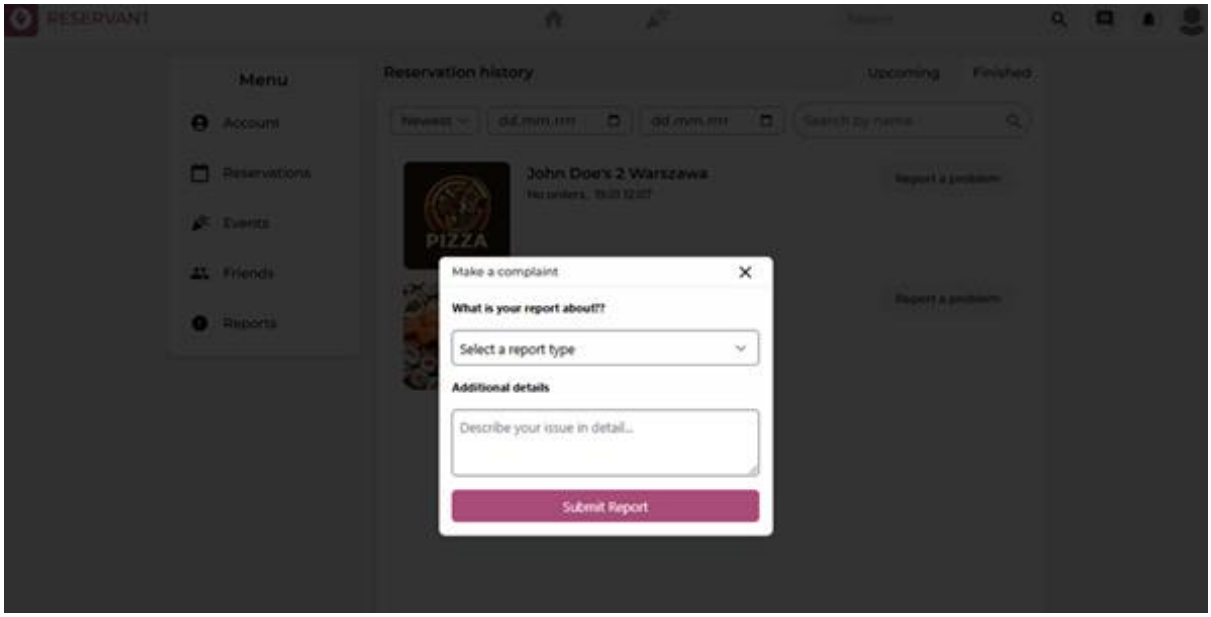
Ekran ten (Rysunek 68) w głównej sekcji konta prezentuje podstawowe dane użytkownika takie jak imię, nazwisko, numer telefonu czy zdjęcie profilowe. Poniżej wspomnianej sekcji znajduje się panel przeznaczony dla transakcji. Klient ma możliwość sprawdzenia historii płatności oraz może doładować konto poprzez kliknięcie przycisku „*Add funds tou your wallet*".

Sekcja *reservation* zgodnie z nazewnictwem przedstawia rezerwacje dokonane przez klienta wraz z wyróżnionymi szczegółami poszczególnych transakcji. W krótkim podsumowaniu zamówienia klient może sprawdzić datę, pozycje z menu przez niego wybrane oraz koszt rezerwacji.



Rysunek 69 Panel historii rezerwacji. Źródło: Opracowanie własne

Dodatkową opcją udostępnioną dla klientów jest „*Report a problem*". Przycisk ten przeznaczony jest do zgłoszenia reklamacji lub poinformowania restauracji o zgubionym przedmiocie w trakcie przebywania w ramach zarezerwowanej wizyty. Następnie wysłane zgłoszenie można sprawdzić w sekcji *Reports*.



Rysunek 70 Panel składania zgłoszeń. Źródło: Opracowanie własne

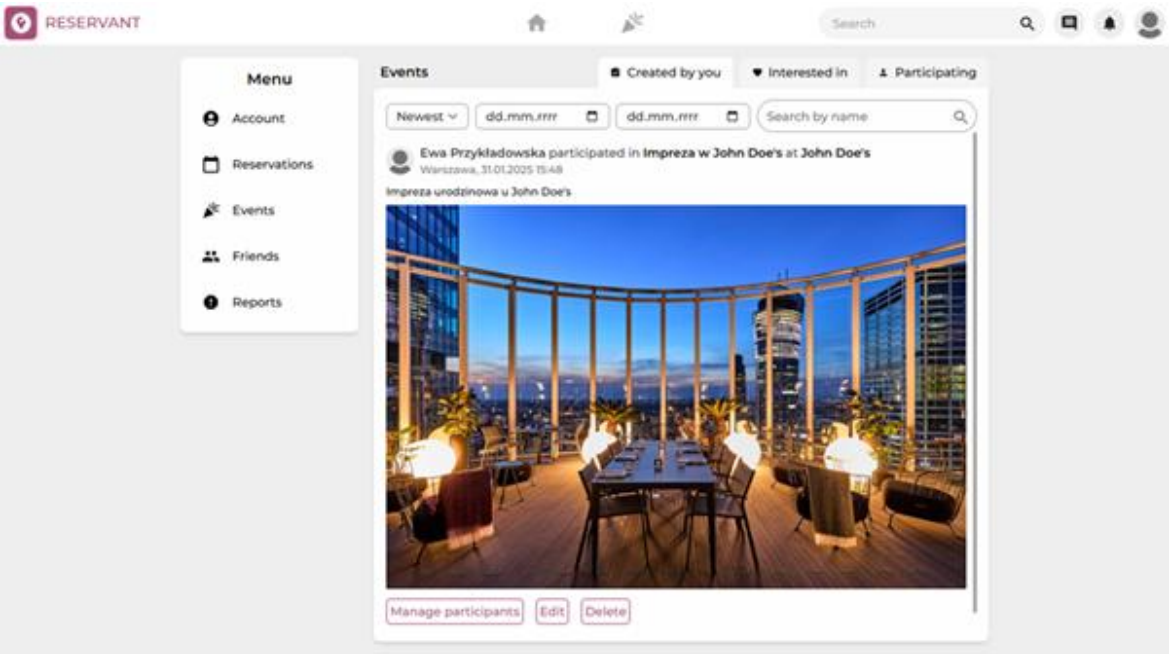
Użytkownicy chcący złożyć zgłoszenie po kliknięciu w przeznaczony do tej funkcjonalności przycisk zobaczą okno modalne, które przedstawia Rysunek 70. Następne kroki które użytkownik musi podjąć to:

1. Wybranie jednego z kilku dostępnych powodów zgłoszenia w liście rozwijanej „*What is your report about*”,
2. Wprowadzenie opisu zdarzenia w dedykowanym ku temu polu tekstowym,
3. Zatwierdzenie zgłoszenia poprzez kliknięcie przycisku „*Submit report*”,
4. Sprawdzenie w sekcji *Reports* czy zgłoszenie pomyślnie zostało wysłane.

Złożone zgłoszenie lub reklamacja następnie trafia do jednego z pracowników biura obsługi klienta, który informuje użytkownika o jego dalszym przebiegu.

Kolejną zakładką w panelu użytkownika jest możliwość zarządzania wydarzeniami (Rysunek 71). Klienci widzą wszystkie wydarzenia powiązane z ich kontem. Wyróżnione są trzy ich rodzaje:

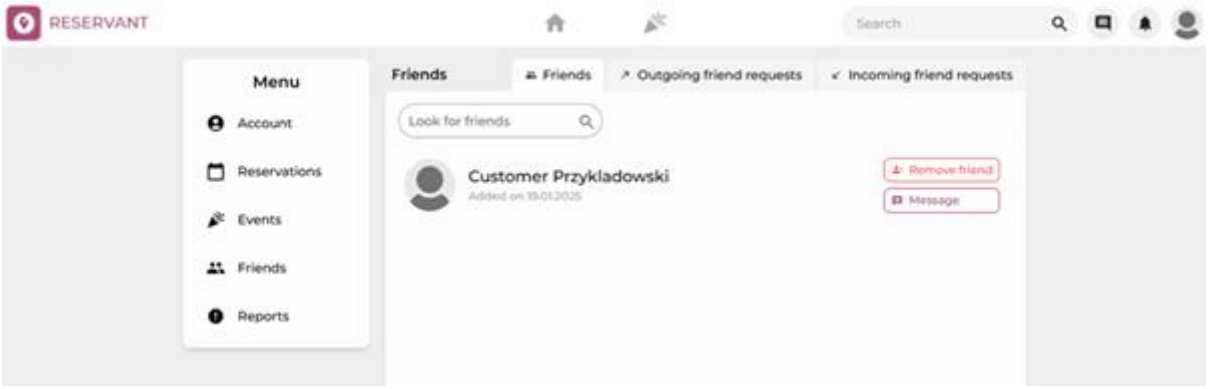
- Utworzone,
- Zainteresowanie,
- Bierzesz udział.



Rysunek 71 Panel zarządzania wydarzeniami. Źródło: Opracowanie własne

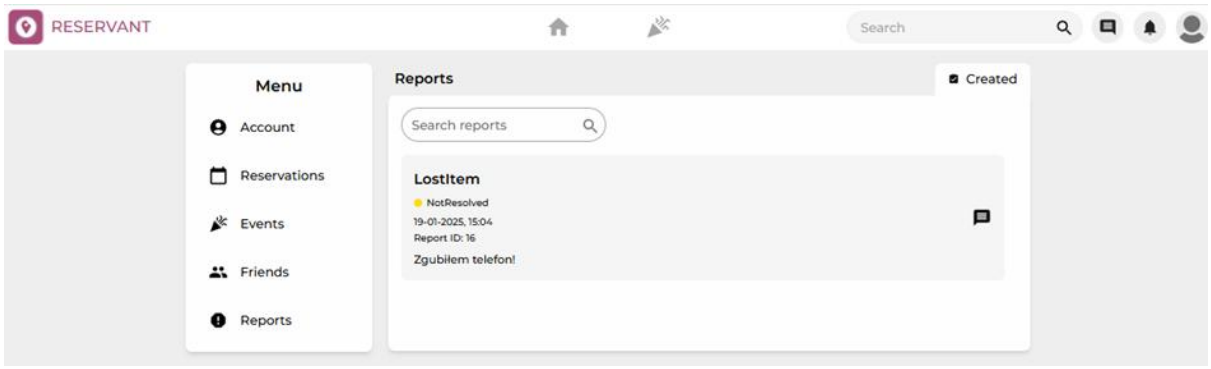
Użytkownicy w każdej chwili mogą zrezygnować z uczestnictwa w wydarzeniu poprzez kliknięcie przycisku „Usuń zainteresowanie". Więcej możliwości otrzymują twórcy wydarzeń, którzy mogą edytować samo wydarzenie poprzez zmianę daty lub lokalu, sprawdzać listę użytkowników biorących udział lub usunąć całe wydarzenie.

Następną zakładką w panelu użytkownika jest sekcja „*Friends*", która przeznaczona jest do sprawdzenia listy znajomych oraz ingerencji w nią poprzez usuwanie poszczególnych użytkowników z listy. Następne sekcje przeznaczone są do zarządzania jak i wglądu zarówno w przychodzące oraz wychodzące zaproszenia do grona znajomych powiązane z kontem zalogowanego użytkownika.



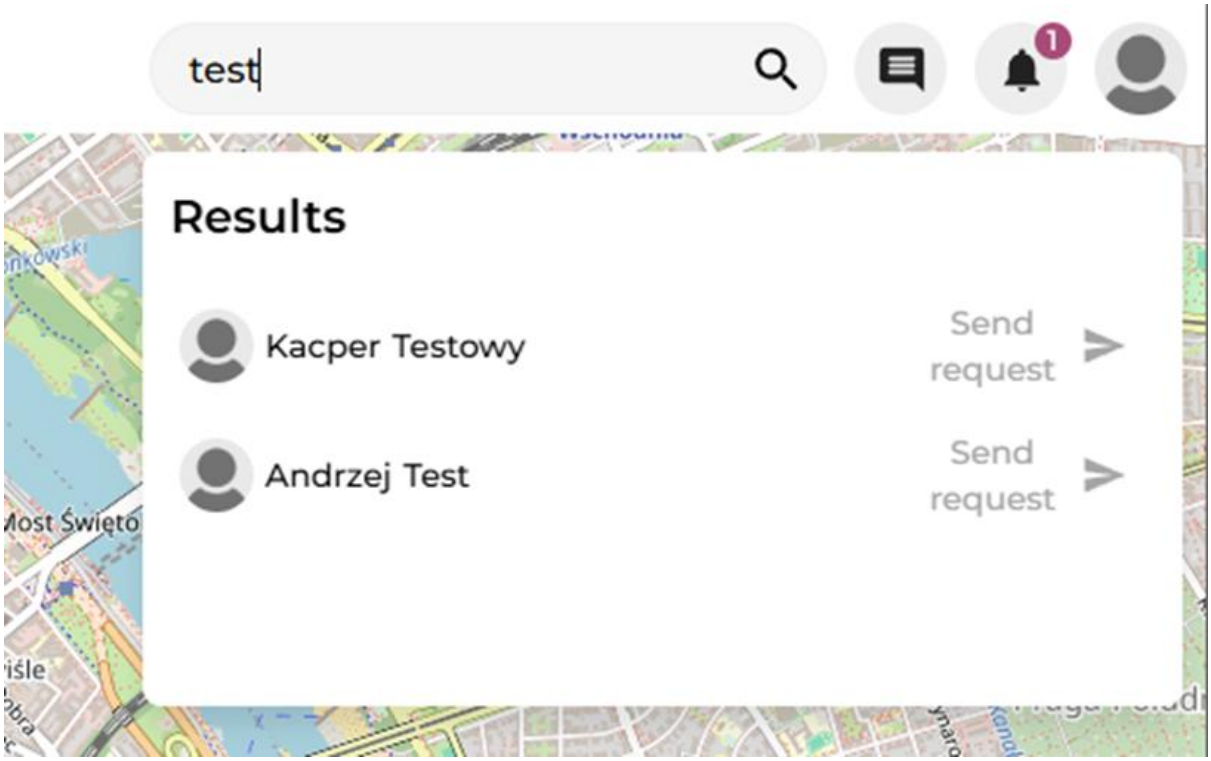
Rysunek 72 Ekran zarządzania znajomymi. Źródło: Opracowanie własne

Ostatnią zakładką jest panel „*Reports*" (Rysunek 73) umożliwia on przeglądanie wysłanych zgłoszeń. Zgłoszenia te mogą dotyczyć zarówno informacji o zagubionym przedmiocie, jak i skarg na zachowanie pracowników. Po wejściu w sekcję „*Reports*" klient ma dostęp do przeglądu zgłoszeń, które wysłał. Prezentowane dane obejmują: datę wysłania, unikalny numer zgłoszenia, jego treść oraz informacje o statusie rozpatrzenia.



Rysunek 73 Ekran zgłoszeń. Źródło: Opracowanie własne

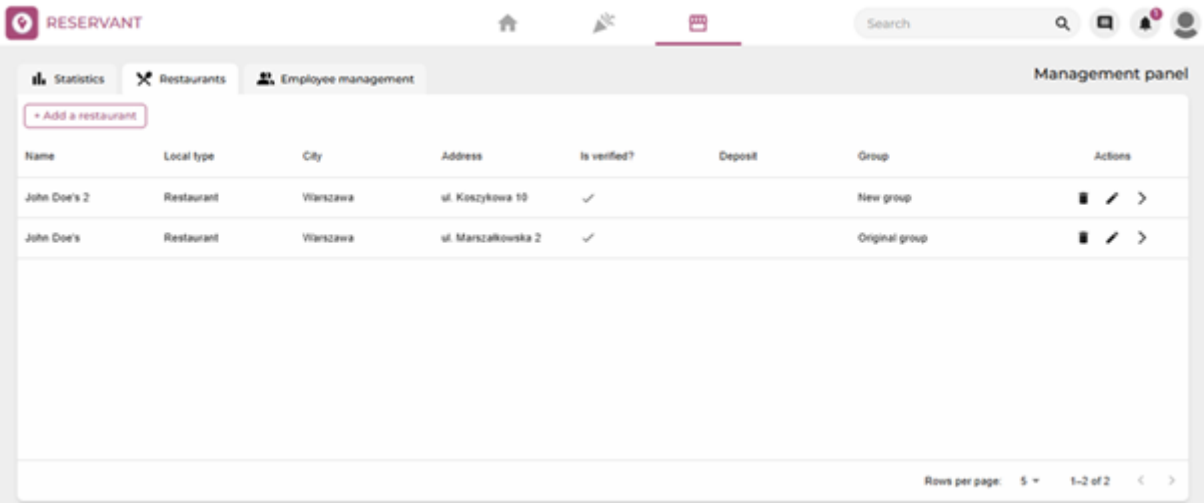
Klienci chcący rozszerzyć grono znajomych mogą wyszukać użytkowników wprowadzając ich imię oraz nazwisko. Przeznaczone ku temu jest pole tekstowe umiejscowione w prawym górnym rogu ekranu z etykietą „*Search for friends*" (Rysunek 74).



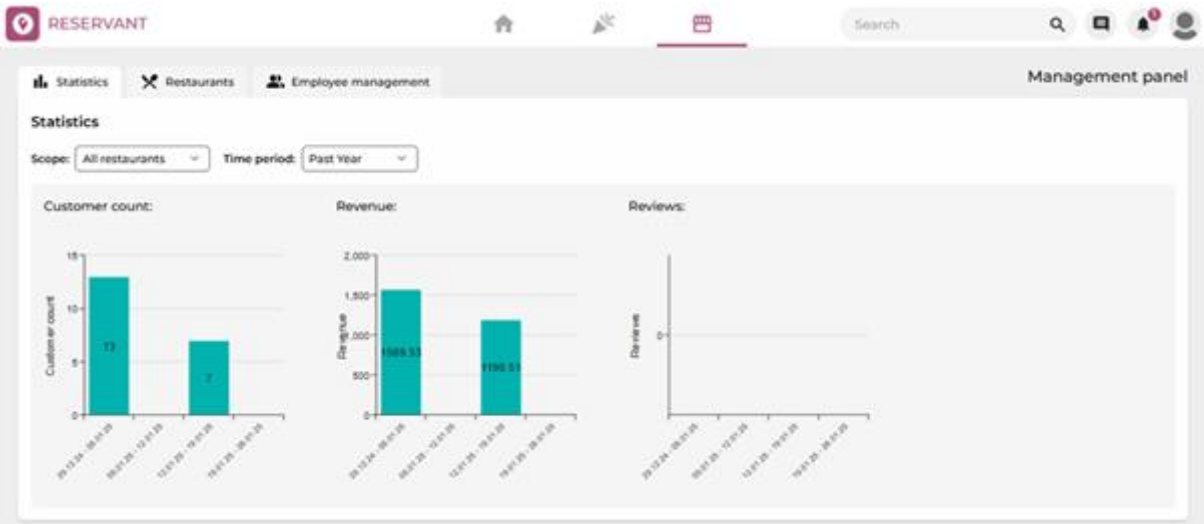
Rysunek 74 Wyszukiwarka użytkowników. Źródło: Opracowanie własne

6.1.7 Widok właściciela restauracji

Właściciel restauracji, po pomyślnym zalogowaniu, ma dostęp do funkcjonalności ściśle związanych z prowadzonymi restauracjami. Po wejściu w panel restauracji widzi trzy sekcje, które odpowiadają kolejno za wszystkie zarządzane przez niego placówki, zatrudnionych pracowników oraz statystyki. Każdy z paneli umożliwia edycję informacji dotyczących restauracji lub pracowników, a także ich ewentualne usunięcie. Dodatkową opcją w sekcji "*Restaurants*" jest możliwość dodania nowej restauracji. Odbywa się to poprzez kliknięcie przycisku "*Add a restaurant*". Po wprowadzeniu wszystkich niezbędnych danych właściciel musi czekać na zatwierdzenie placówki jako zweryfikowana przez Biuro obsługi klienta. Przykładowy ekran przedstawia Rysunek 75 oraz Rysunek 76.



Rysunek 75 Ekran listy prowadzonych restauracji. Źródło: Opracowanie własne

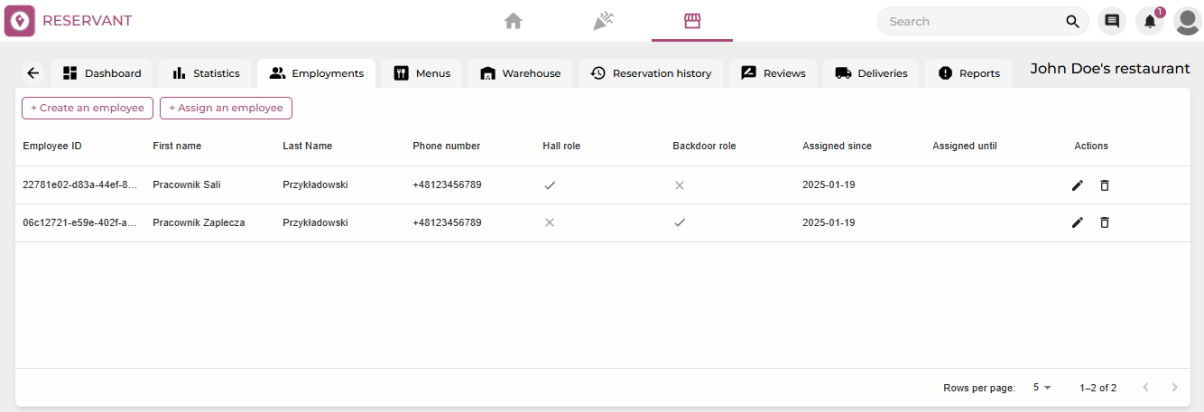


Rysunek 76 Panel statystyk restauracji. Źródło: Opracowanie własne

Właściciel restauracji może następnie wejść w szczegóły wybranego lokalu, gdzie wyświetlane są wszystkie związane z nim informacje. W górnej części ekranu znajduje się pasek nawigacyjny z zakładkami, umożliwiający przełączanie między różnymi sekcjami systemu, takimi jak *dashboard*, *statistics*, *employments*, *menus*, *warehouse*, *reservation history*, *reviews*, *deliveries* oraz *reports*.

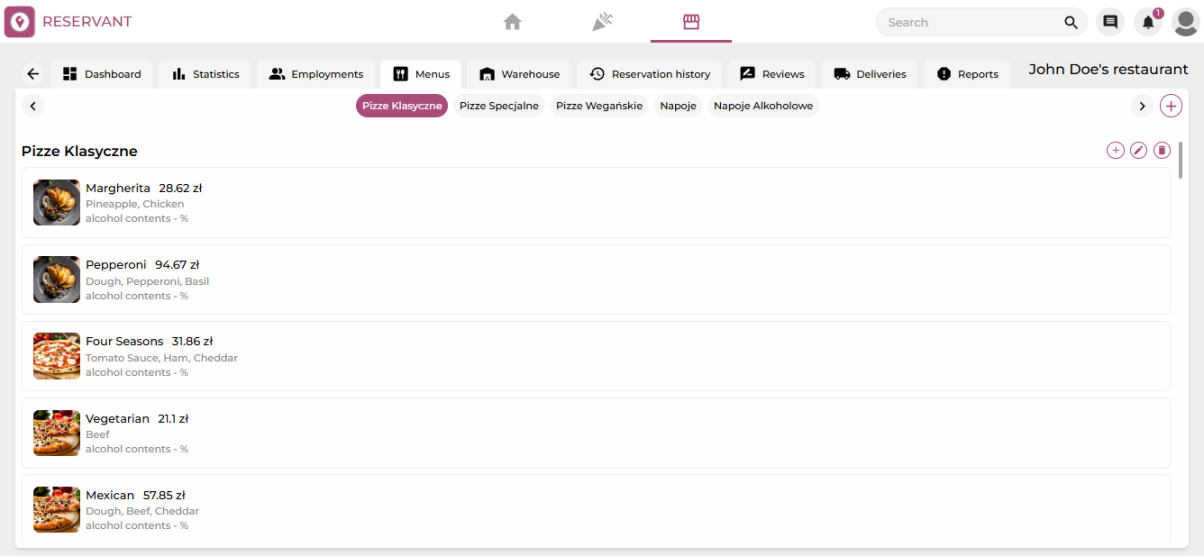
"*Dashboard*" prezentuje informacje ogólne o restauracji. Są to dane podawane w trakcie rejestracji. Sekcja „statistics" zgodnie z nazewnictwem wyświetla statystyki danej restauracji takie jak ilość klientów, przychód oraz wystawione opinie.

W panelu „*Employments*" (Rysunek 77) właściciel może przypisać dostępnych pracowników poprzez przycisk "assign an employee" do restauracji lub usunąć istniejącego pracownika. Dodatkowo, klikając przycisk „Add an employee", może utworzyć nowego pracownika, który nie znajduje się w systemie.

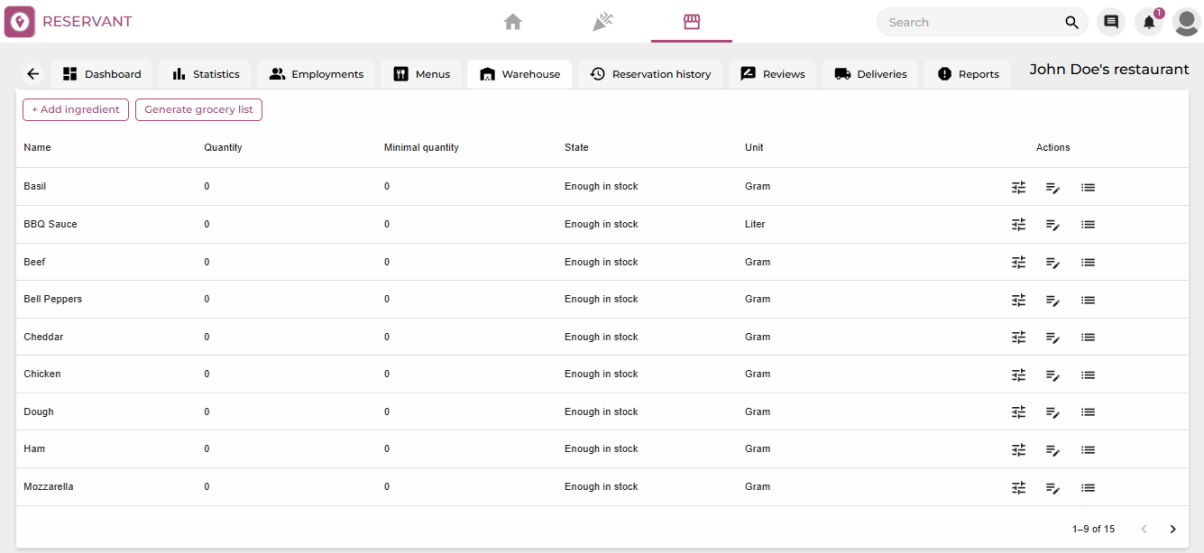


Rysunek 77 Ekran listy pracowników w danej restauracji. Źródło: Opracowanie własne

Sekcja „*Menus*" umożliwia zarządzanie menu restauracji. Właściciel restauracji może dodawać, edytować oraz usuwać poszczególne pozycje w menu. Podczas dodawania nowej pozycji istotne jest uwzględnienie składników potrzebnych do przygotowania dania, co jest bezpośrednio powiązane z panelem „*Warehouse*". Po każdym zamówieniu ilość składników w magazynie jest automatycznie aktualizowana przez co pracownicy mogą w łatwy sposób kontrolować stan zapasów. W przypadku braków właściciel może utworzyć listę zakupów, aby uzupełnić zapasy magazynowe. Przykładowy ekran przedstawia Rysunek 78 oraz Rysunek 79 .

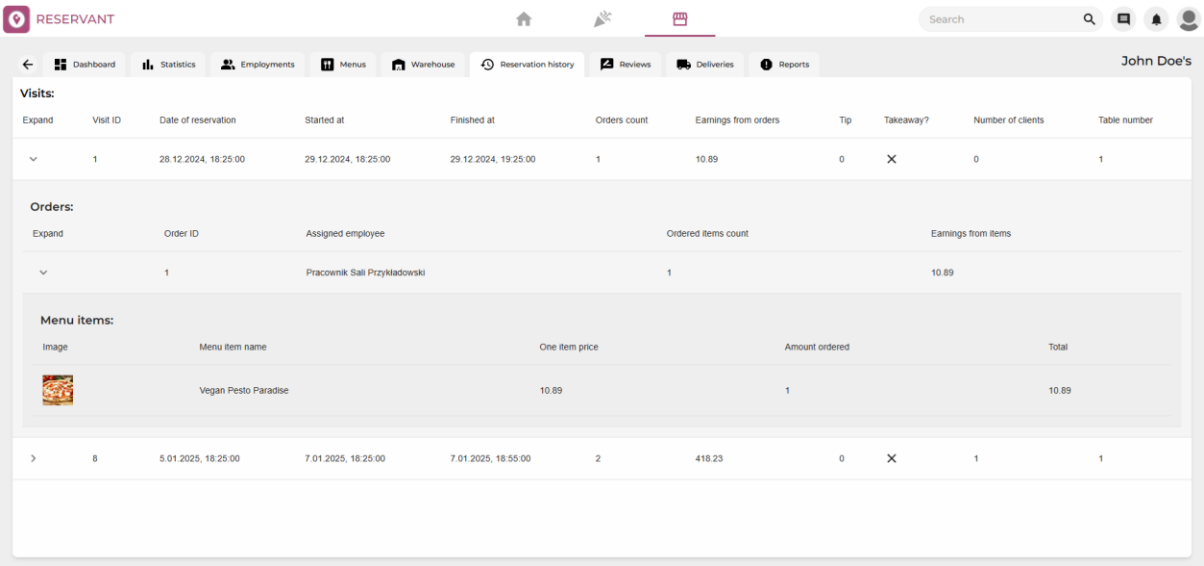


Rysunek 78 Ekran edycji menu. Źródło: Opracowanie własne



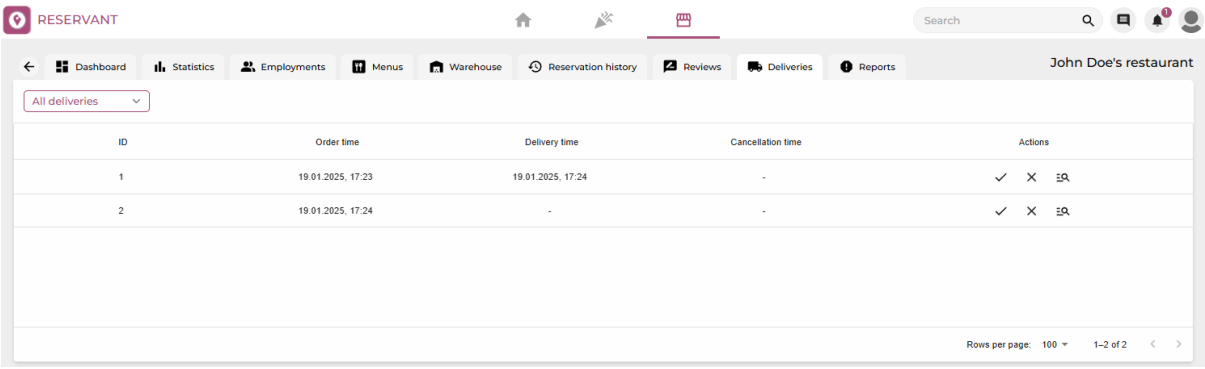
Rysunek 79 Ekran stanu magazynu. Źródło: Opracowanie własne

W zakładce "*Reservation history*" można znaleźć szczegółowe informacje o zamówieniach jakie obsłużyła dana restauracja. Rysunek 80 prezentuje informację o jednej z wizyt. Widoczne jest tam m.in. co zostało zamówione, kto był klientem, który pracownik obsłużył klienta i koszt zamówienia.



Rysunek 80 Ekran historii rezerwacji. Źródło: Opracowanie własne

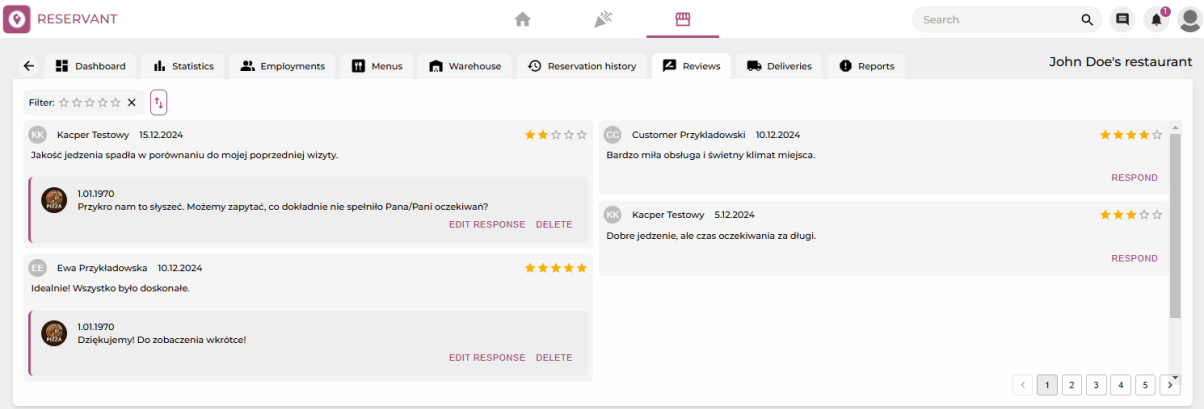
Zakładka "*Deliveries*" (Rysunek 81) pozwala na sprawdzenie nadchodzących dostaw oraz tych które już się odbyły. Po odebraniu zamówienia pracownik oznacza dostawę jako zrealizowaną.



Rysunek 81 Ekran dostaw do restauracji. Źródło: Opracowanie własne

Panel „*Reservation history*" służy do archiwizowania zakończonych rezerwacji. Przechowywane dane obejmują datę rezerwacji, datę rozpoczęcia i zakończenia wizyty, informacje o ewentualnej płatności, numer stolika oraz liczbę gości przypisanych do rezerwacji.

Właściciel restauracji może również przeglądać wszelkie opinie wystawione dla swojego lokalu. Znajdują się one w zakładce "*Reviews*" (Rysunek 82). Klikając w przycisk "*respond*" można dodać odpowiedź lokalu do wystawionej opinii. Zamieszczona odpowiedź może zostać edytowana lub usunięta.



Rysunek 82 Ekran opinii lokalu. Źródło: Opracowanie własne

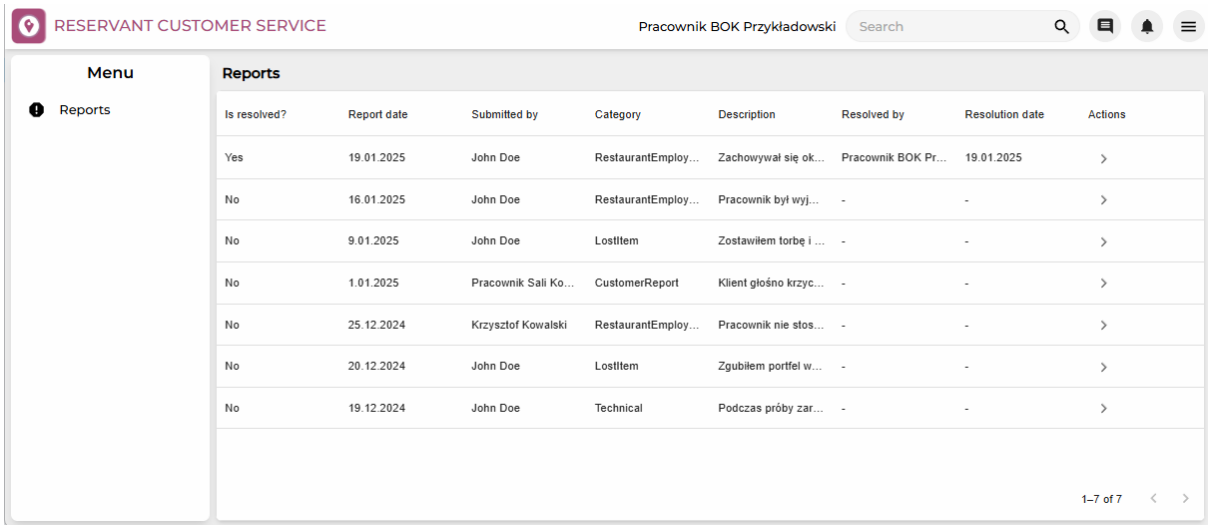
Ostatnia zakładka *Reports* odnosi się do wszelkich zgłoszeń związanych z lokalem. Mogą one odnosić się zarówno do zgubionych przedmiotów jak i skarg na zachowanie danego pracownika. Zgłoszenia te są rozpatrywane przez Biuro obsługi klienta zatem ekran dla właściciela lokalu jest tylko poglądowy.

6.1.8 Panel biura obsługi klienta

Po zalogowaniu pracownik biura obsługi klienta zostaje przeniesiony na ekran związany ze zgłoszeniami lub reklamacjami składanymi przez klientów. Rysunek 83 prezentuje wspomniany panel. Składa się on z głównej sekcji "*Reports*", w której znajdują się wszystkie złożone zgłoszenia. Pracownik biura obsługi klienta ma dostęp do następujących informacji:

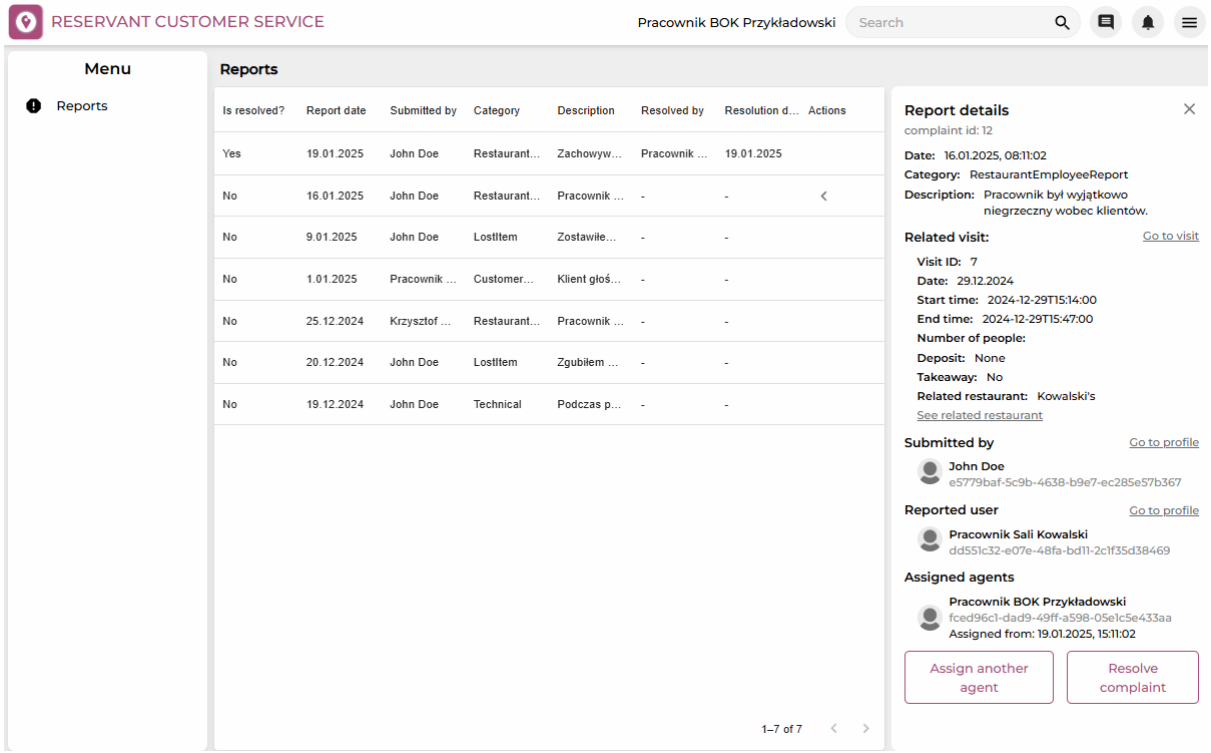
- Status zgłoszenia (czy zostało rozwiązane),
- Data złożenia zgłoszenia,
- Osoba zgłaszająca,
- Kategoria zgłoszenia,

- Opis zgłoszenia,
- Osoba rozwiązująca zgłoszenie,
- Data rozwiązania zgłoszenia.



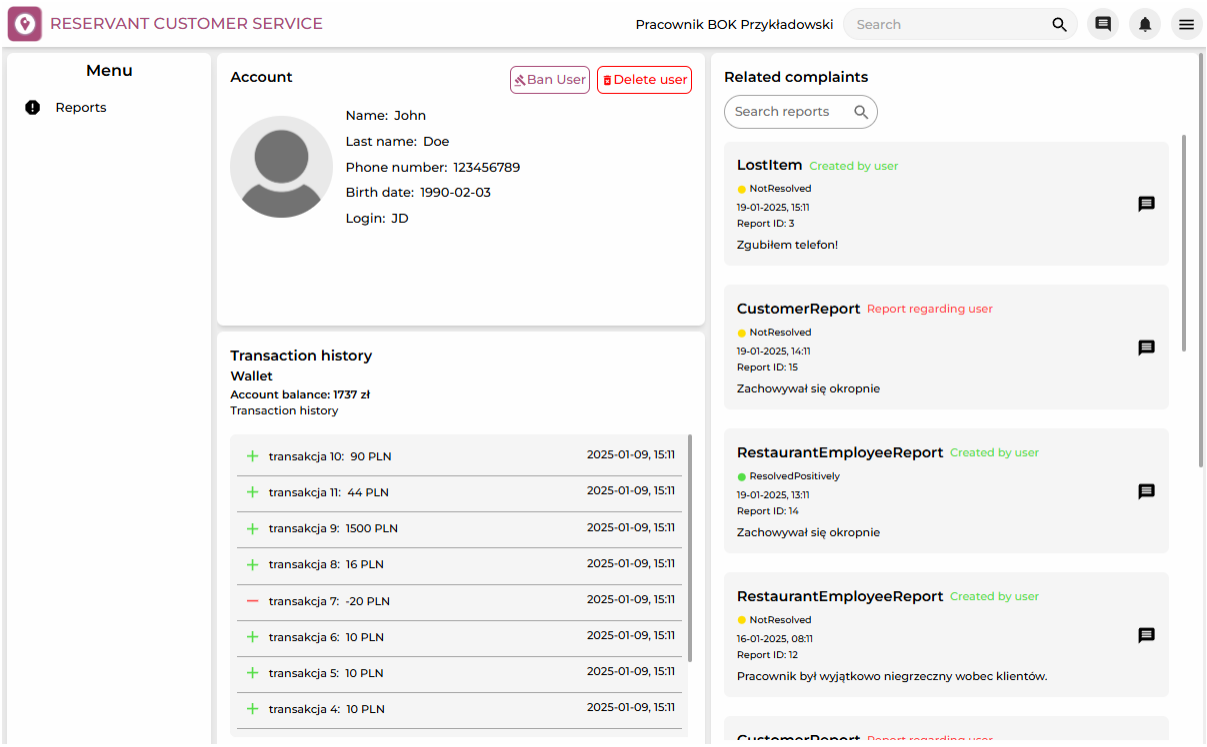
Rysunek 83 Ekran główny biura obsługi klienta. Źródło: Opracowanie własne

Po kliknięciu w strzałkę '*Actions'* pracownik widzi szczegółowe informacje dotyczące danego zgłoszenia, tak jak prezentuje to Rysunek 84.



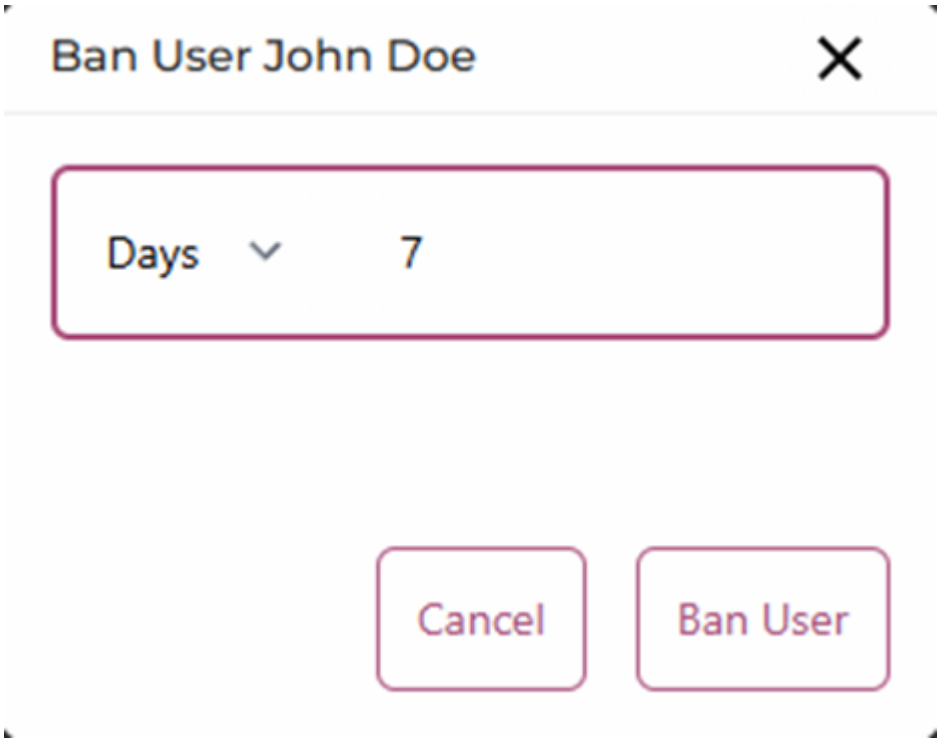
Rysunek 84 Ekran szczegóły zgłoszenia. Źródło: Opracowanie własne

Pracownik może sprawdzić historię klienta, który złożył zgłoszenie poprzez kliknięcie przycisku '*Go to profile*' lub informacje odnośnie do wizyty poprzez przycisk '*Go to visit*'.



Rysunek 85 Szczegóły konta klienta składającego zgłoszenie. Źródło: Opracowanie własne

W szczegółowym widoku klienta pracownik może nałożyć na użytkownika czasowe wykluczenie (*ban*), klikając przycisk "*Ban User*". Okres wykluczenia jest ustalany według decyzji agenta biura obsługi klienta. Proces kończy się zatwierdzeniem poprzez ponowne kliknięcie przycisku "*Ban User*". Wspomnianą funkcjonalność prezentuje Rysunek 85 oraz Rysunek 86.

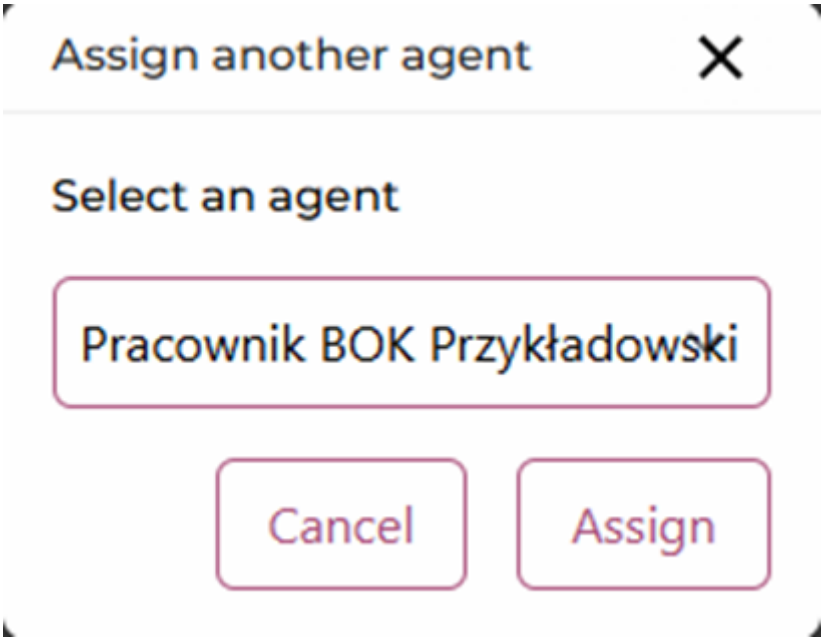


Rysunek 86 Ekran wykluczenia użytkownika z systemu. Źródło: Opracowanie własne

Po zapoznaniu się ze zgłoszeniem, pracownik może zakończyć jego rozpatrywanie, klikając przycisk "*Resolve complaint*". Następnie wprowadza informację zwrotną dla klienta i kończy proces, wybierając przycisk "*Resolve*" lub "*Reject*" (Rysunek 87).

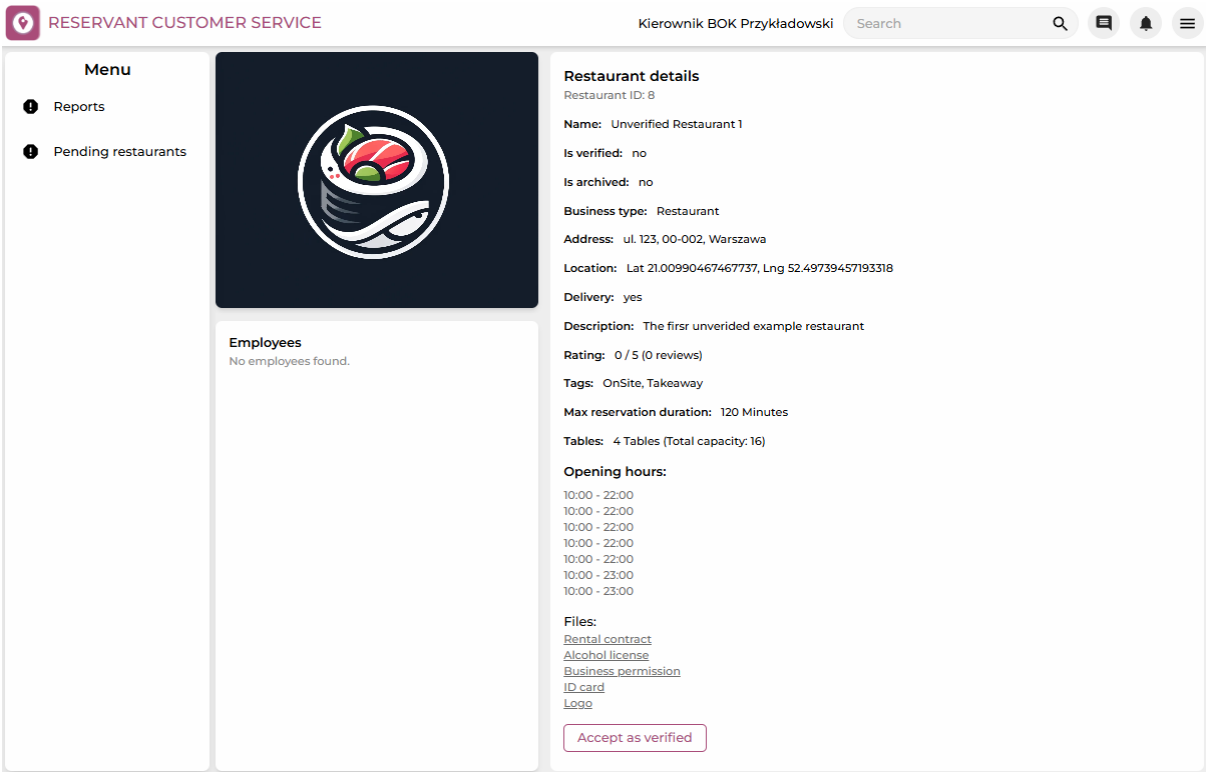
Rysunek 87 Ekran rozwiązywania zgłoszenia. Źródło: Opracowanie własne

Pracownik może również zdecydować się na przypisanie zgłoszenia do innego agenta. W tym celu należy kliknąć przycisk "*Assign to another agent*", wybrać agenta z listy i zatwierdzić wybór, klikając przycisk "*Assign*" (Rysunek 88).



Rysunek 88 Ekran przypisania zgłoszenia do agenta. Źródło: Opracowanie własne

Użytkownik przypisany jako Manager biura obsługi klienta może dodatkowo zatwierdzać zgłoszenia dotyczące weryfikacji restauracji. Do tego celu służy panel „*Pending restaurants*" zaprezentowany przez Rysunek 89. Po wejściu w szczegóły zgłoszenia pracownik ma dostęp do szczegółowych informacji o restauracji oraz załączników dodanych przez jej właściciela. Proces weryfikacji kończy się kliknięciem przycisku „*Accept as verified*".



Rysunek 89 Ekran zatwierdzenia restauracji. Źródło: Opracowanie własne

6.1.9 Dodatkowe funkcje aplikacji webowej

Aplikacja webowa systemu Reservant wyposażona została w opcję zmiany motywu kolorystycznego. Domyślnym trybem jest motyw jasny w którym dominuje kolor biały. Alternatywnym trybem jest motyw ciemny „*Dark mode*”. Charakteryzuje się on ciemną kolorystyką z ciemnozielonymi akcentami. Tryb ten nie zmienia w żadnym stopniu funkcjonalności aplikacji, udostępnia jedynie dodatkową możliwość dostosowania strony do preferencji użytkownika.

Kolejną opcją udostępnioną dla użytkowników jest wybór języka w jakim aplikacja będzie prezentowana. Klient może wybrać pomiędzy językiem polskim lub angielskim. Aplikacja domyślnie prezentowana jest w języku angielskim zatem opcja ta kryje się pod przyciskiem „*Language*”.

6.2 Aplikacja mobilna

W poniższych podrozdziałach zaprezentowano kluczowe funkcjonalności aplikacji mobilnej systemu Reservant. Szczególną uwagę poświęcono różnicom w interfejsie użytkownika, wynikającym z przypisanej roli w systemie.

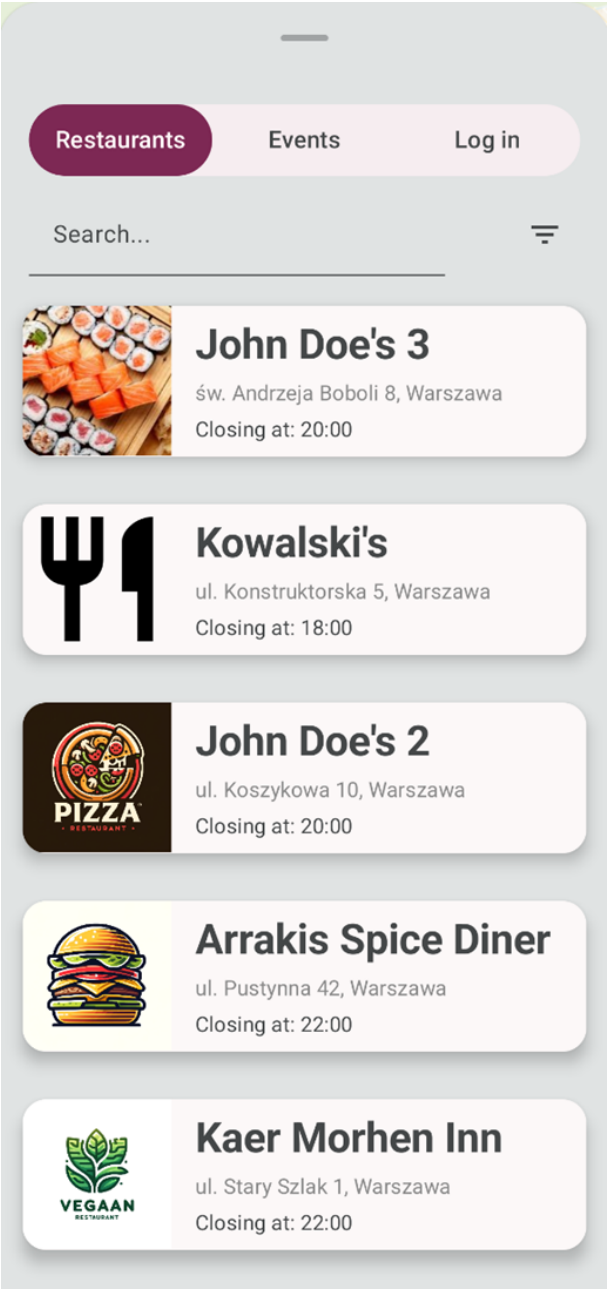
6.2.1 Ekran startowy gościa

Po otwarciu niniejszej aplikacji na smartfonie, wyświetla się mapa służąca do przeglądania lokali w pobliżu aktualnej lokalizacji użytkownika, co ukazuje Rysunek 90.

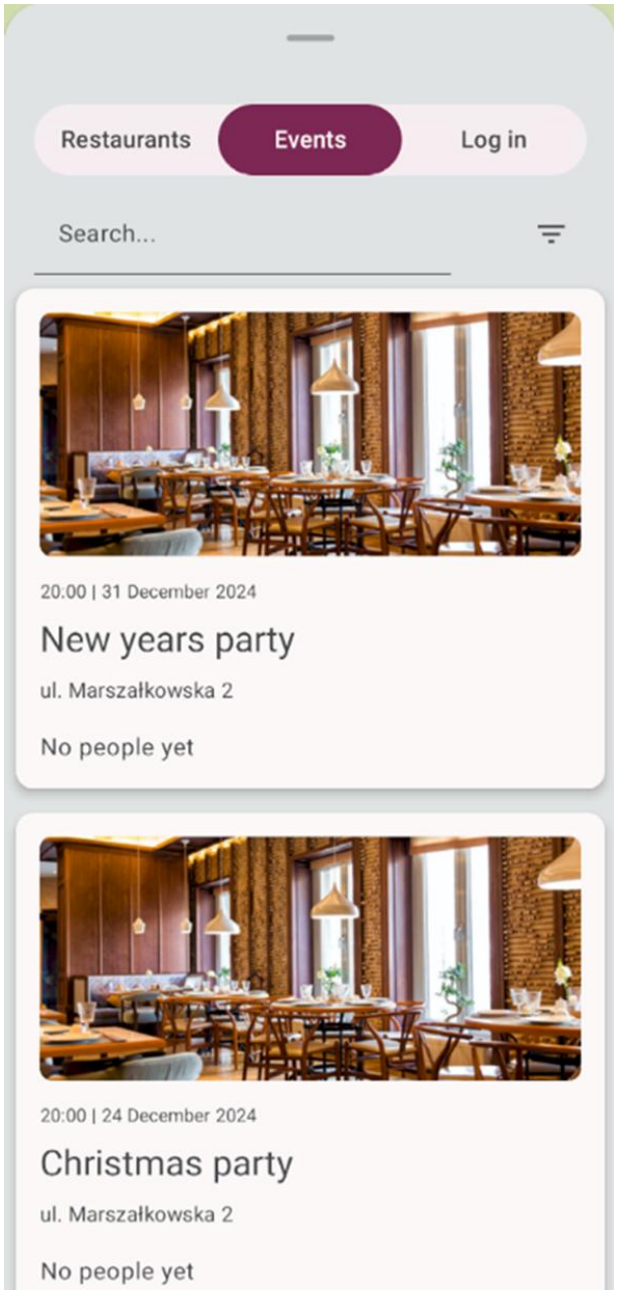
Rysunek 90 Strona główna aplikacji. Źródło: Opracowanie własne

Wysuwając panel znajdujący się na dole ekranu, użytkownikowi ukazuje się ekran z następującymi zakładkami "*Restaurants*", "*Events*" oraz "*Log in*". Domyślnie wybrana jest zakładka restauracji, na której użytkownik może przeglądać listę restauracji. Dla konkretnej restauracji wyświetlana jest jej nazwa, lokalizacja oraz godzina zamknięcia (Rysunek 91).

Po zmianie zakładki na "*Events*" wyświetla się lista aktualnych eventów, które będą odbywać się dla dostępnych w systemie restauracji (Rysunek 92). Dla wydarzenia można zobaczyć jego nazwę, lokalizację, datę i godzinę rozpoczęcia, a także czy jakiś użytkownik aplikacji zadeklarował udział w evencie. W przypadku eventu, dla którego nikt jeszcze nie zgłosił udziału wyświetlamy informację "*No people yet*".



Rysunek 91 Ekran lokali i restauracji. Źródło: Opracowanie własne



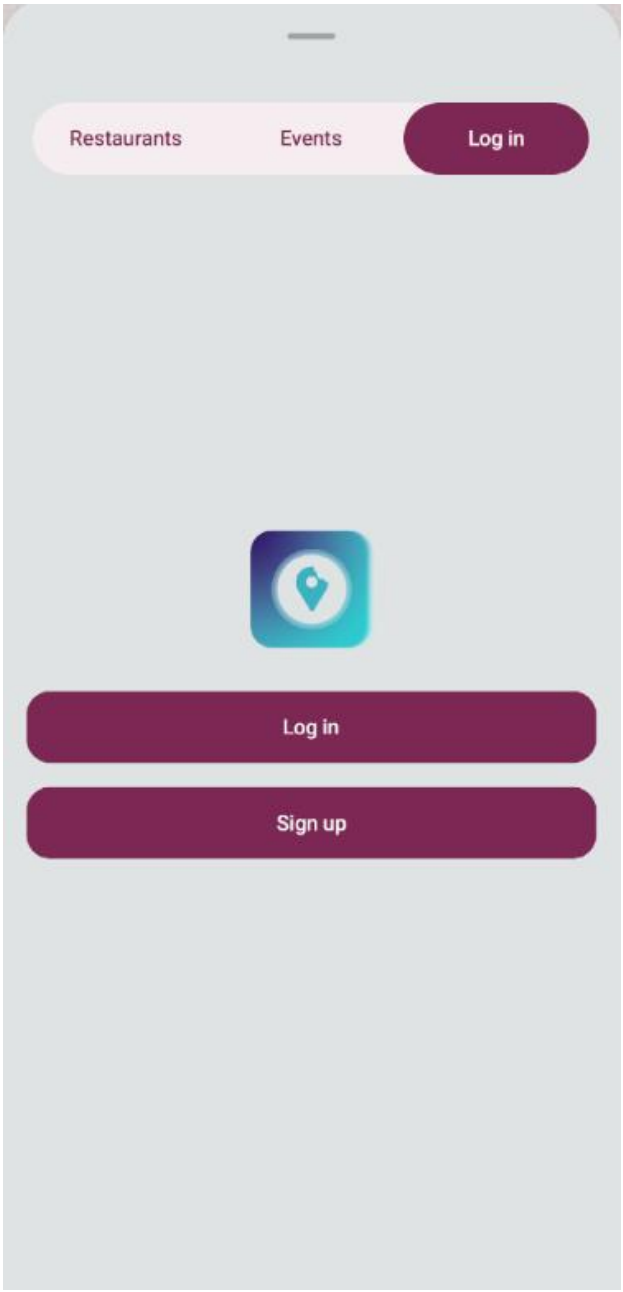
Rysunek 92 Ekran wydarzeń. Źródło: Opracowanie własne

6.2.2 Logowanie/Rejestracja

Przechodząc do zakładki "*Log in*" (Rysunek 93) użytkownik może zalogować się do aplikacji za pomocą przycisku "*Log in*" w przypadku, gdy posiada konto w aplikacji jako:

- Klient
- Właściciel lub pracownik restauracji
- Kierownik lub pracownik BOK

Przy użyciu przycisku "*Sign up*" użytkownik Gość może zarejestrować się w systemie.

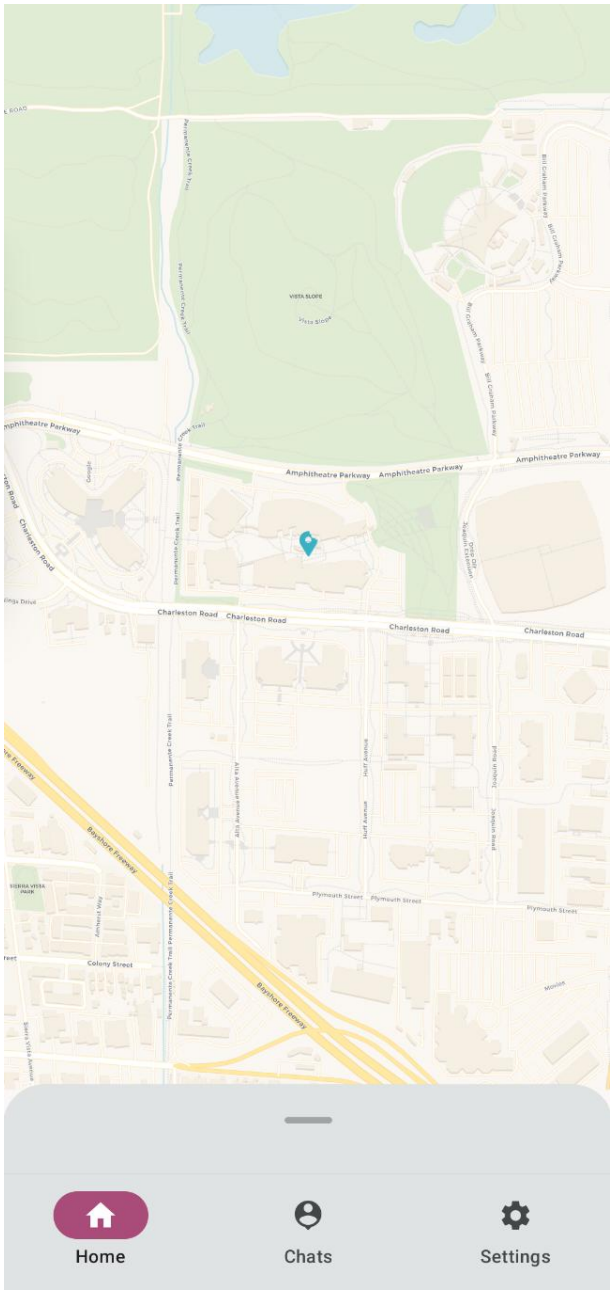


Rysunek 93 Logowanie do aplikacji mobilnej. Źródło: Opracowanie własne

6.2.3 Widok zalogowanego użytkownika

Po zalogowaniu się do aplikacji jako Klient, na dole ekranu wyświetla się dodatkowy pasek akcji. Dostępne zakładki to:

- "*Home*" domyślna zakładka, ekran początkowy aplikacji wyświetlający mapę oraz możliwy do wysunięcia z dołu panel aktywności zawierający restauracje i eventy (Rysunek 94);
- "*Chats*" zakładka, w której użytkownik może prowadzić konwersację z innymi użytkownikami aplikacji (Rysunek 95);
- "*Settings*" zakładka z ustawieniami oraz informacjami o użytkowniku. Szczegóły dotyczące dostępnych akcji w ramach tej zakładki, zostaną zaprezentowane w podrozdziale numer 6.2.6.



Rysunek 94 Ekran startowy zalogowanego użytkownika, Źródło: Opracowanie własne

Rysunek 95 Ekran wiadomości. Źródło: Opracowanie własne

6.2.4 Ekran restauracji

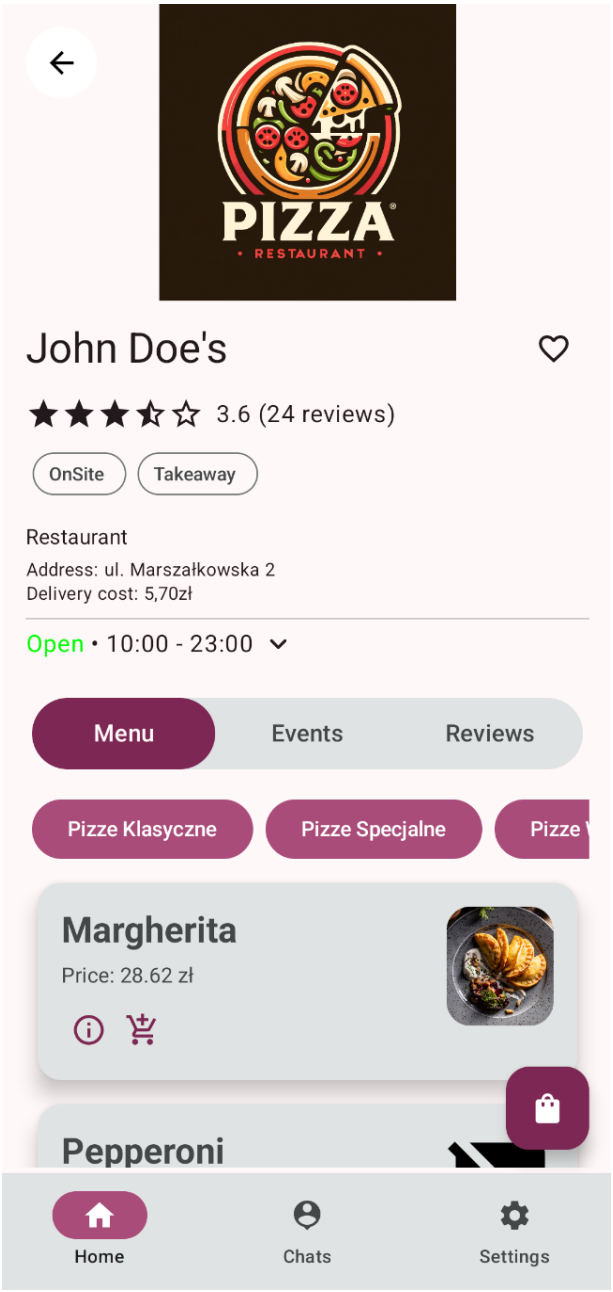
Po kliknięciu w wybraną restaurację wyświetlany jest ekran, na którym znajduje się szereg dostępnych aktywności (Rysunek 96).

W górnym panelu oprócz zdjęcia i loga restauracji wyświetlane są dane techniczne restauracji:

- Ocena lokalu
- Informacje o opcjach dostawy: “*OnSite*”, “*Takeaway*”, “*Delivery*”
- Adres
- Cena dostawy
- Godziny otwarcia oraz obecny status otwarcia (“*Open*” lub “*Closed*”)

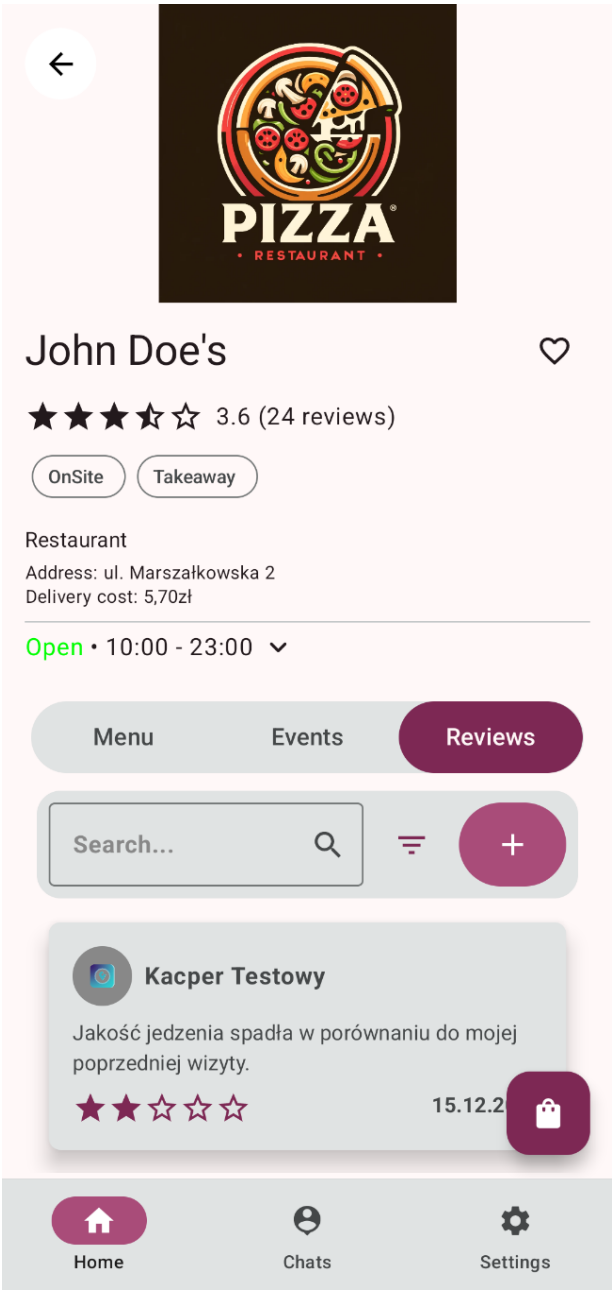
W panelu znajdującym się poniżej danych technicznych, Klient może przeglądać “Menu” restauracji z podziałem na kategorie np. “Pizze Klasyczne”, “Pizze Specjalne”, “Pizze Wegańskie”, “Napoje”, “Napoje alkoholowe” jak na zaprezentowanym rysunku.

W zakładce “Events” wyświetlane są wydarzenia (wyglądające adekwatnie tak jak pokazuje to Rysunek 92) organizowane przez wybraną restaurację.

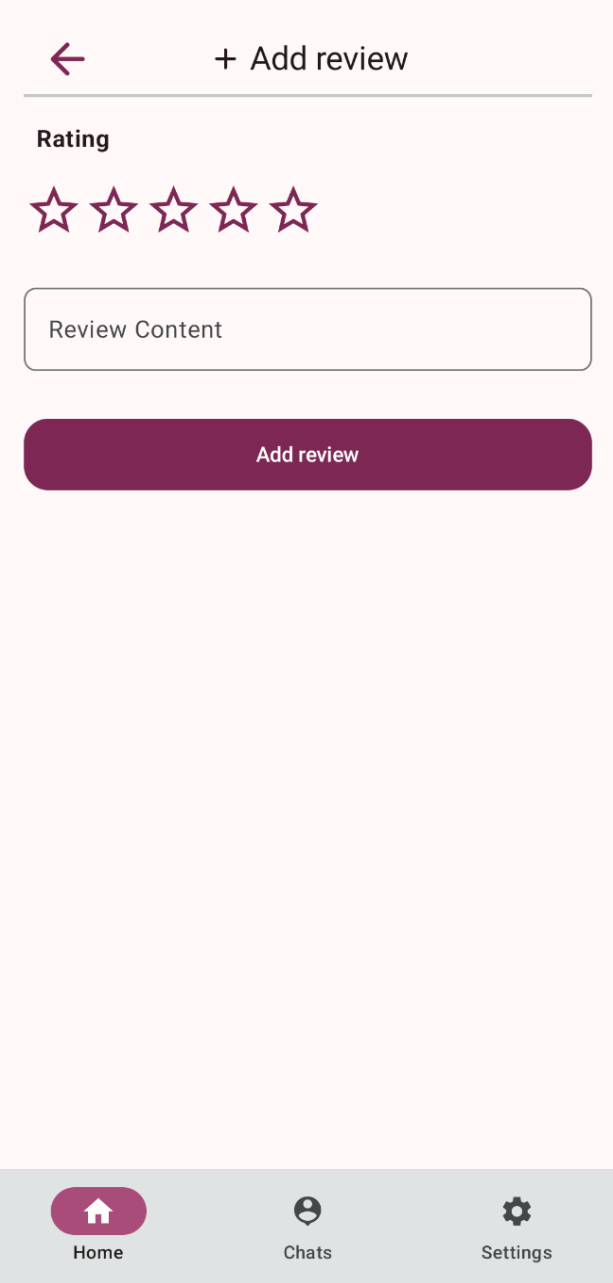


Rysunek 96 Ekran restauracji. Źródło: Opracowanie własne

Po wyborze zakładki “*Reviews*” (Rysunek 97) użytkownik może wyszukiwać i przeglądać komentarze z ocenami dotyczące restauracji. Klient może też samodzielnie wystawić opinię restauracji za pomocą “+” znajdującego się po prawej stronie, pod nagłówkiem “*Reviews*”. Wyświetlany jest formularz oceny, który pokazuje Rysunek 98. Użytkownik wprowadza ocenę za pomocą gwiazdek, wpisuje komentarz i publikuje ocenę za pomocą przycisku “*Add review*”.



Rysunek 97 Zakładka reviews dla restauracji. Źródło: Opracowanie własne



Rysunek 98 Formularz dodawania oceny dla restauracji. Źródło: Opracowanie własne

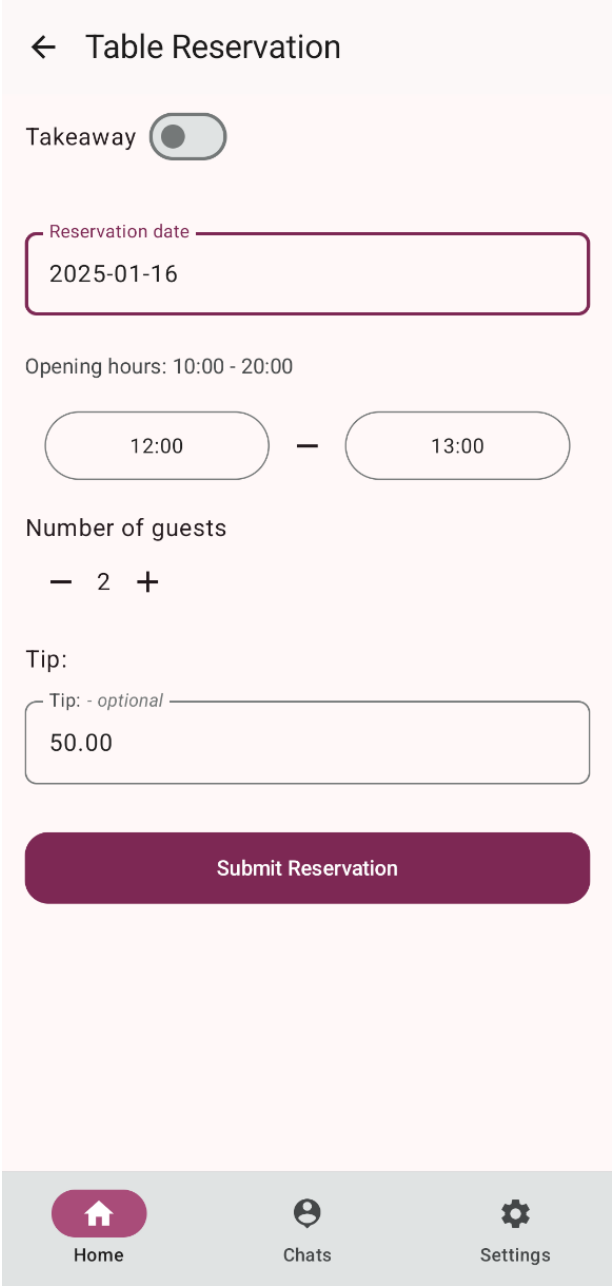
6.2.5 Złożenie zamówienia oraz rezerwacji

Proces złożenia zamówienia bądź rezerwacji stolika w wybranej restauracji, rozpoczyna się od przejścia do ekranu głównego restauracji.

W celu złożenia rezerwacji na stolik w lokalu, należy kliknąć przycisk “Reserve a table”. Użytkownikowi wyświetli się ekran, na którym wybiera:

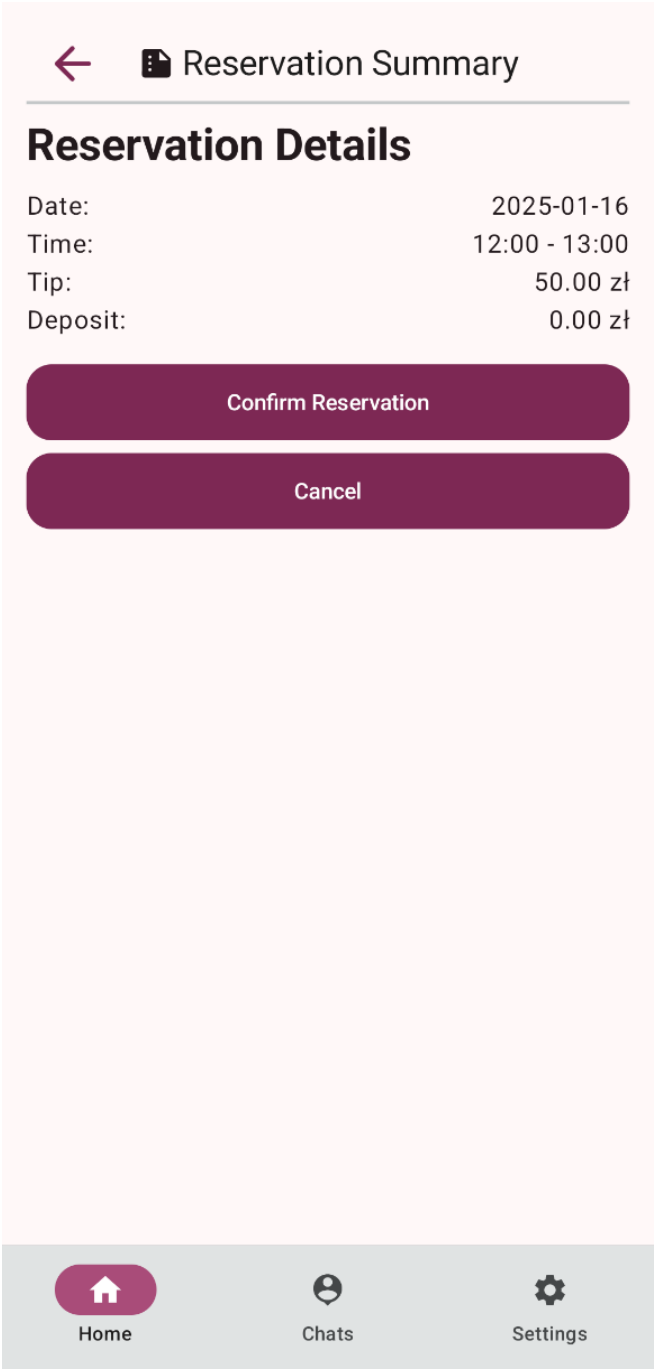
1. datę rezerwacji,
2. godzinę rezerwacji,
3. liczbę gości,
4. opcjonalny napiwek.

Ekran rezerwacji został zaprezentowany poprzez Rysunek 99.



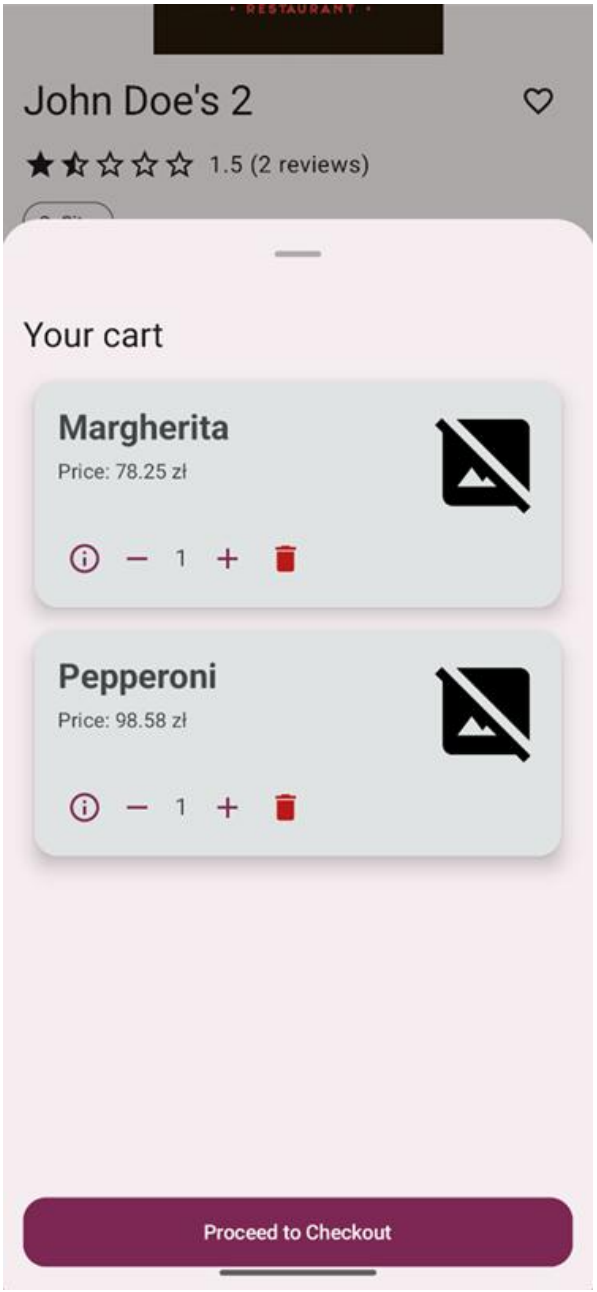
Rysunek 99 Ekran rezerwacji. Źródło: Opracowanie własne

Po wypełnieniu wyżej wymienionych pól, za pomocą "*Submit Reservation*" użytkownik przechodzi do podsumowania rezerwacji, które prezentuje Rysunek 100. Po potwierdzeniu rezerwacji za pomocą przycisku "Confirm Reservation" wyświetli się pop up informujący o utworzeniu rezerwacji.



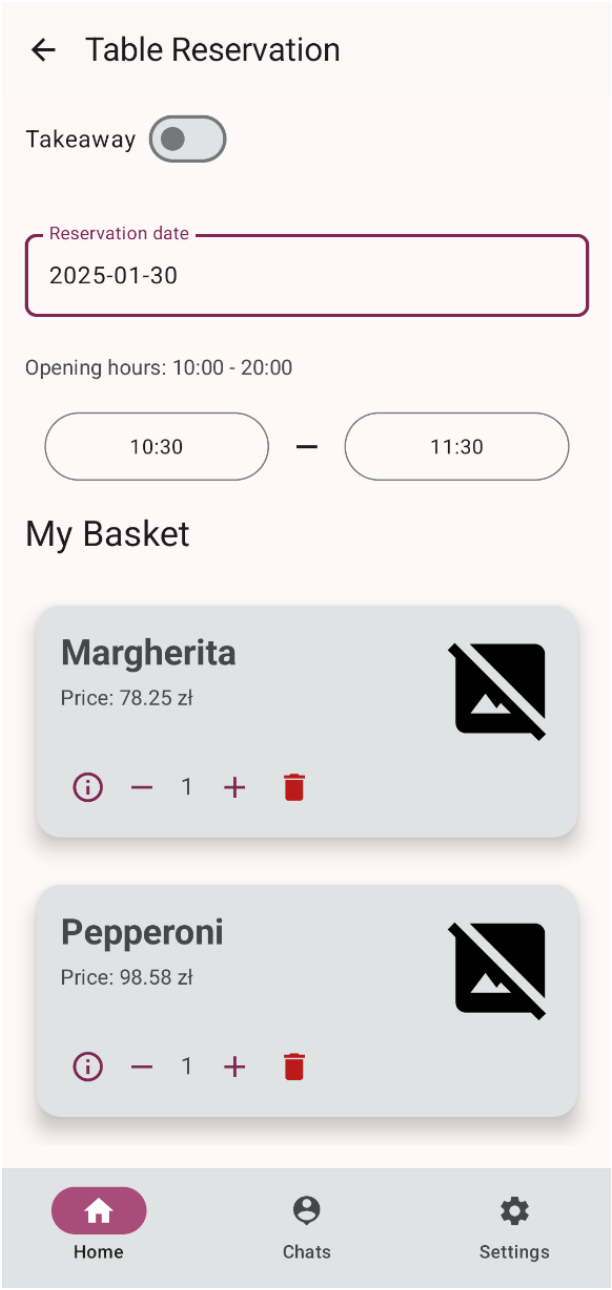
Rysunek 100 Podsumowanie rezerwacji. Źródło: Opracowanie własne

Z poziomu ekranu głównego restauracji, użytkownik ma możliwość złożenia zamówienia na wybrane pozycje z menu, na miejscu lub na wynos. Klient przegląda menu i dodaje do koszyka interesujące go dania i napoje, za pomocą ikony (koszyka sklepowego z plusem) znajdującej się przy wybranej pozycji z menu. Po dodaniu wybranych produktów i w celu złożenia zamówienia na wynos lub zamówienia z rezerwacją, Klient przechodzi do koszyka znajdującego się w prawym dolnym rogu ekranu i wybiera opcję "*Proceed to Checkout*", jak przedstawia Rysunek 101



Rysunek 101 Formularz składania rezerwacji. Źródło: Opracowanie własne

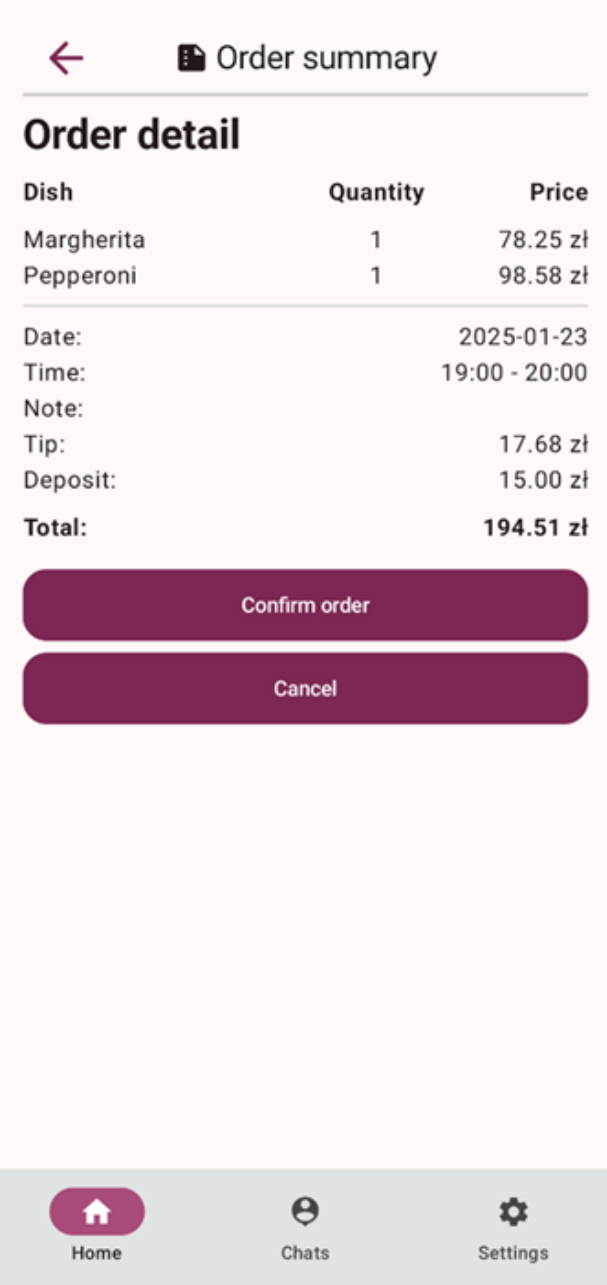
Po kliknięciu wspomnianego przycisku, Klientowi wyświetla się formularz złożenia zamówienia na wynos bądź zamówienia z rezerwacją. Dla opcji złożenia zamówienia z rezerwacją wyświetla się następujący ekran (Rysunek 102). W sekcji "*My Basket*" znajdują się wybrane przez Klienta pozycje z menu.



Rysunek 102 Ekran zamówienia z rezerwacją. Źródło: Opracowanie własne

Po przesunięciu formularza, Klient może dodać opcjonalny napiwek oraz zostawić opcjonalną wiadomość do zamówienia. Na dole ekranu wyświetlany jest całkowity koszt zamówienia.

W celu przejścia do podsumowania rezerwacji Klient wybiera przycisk "*Submit order*" znajdujący się na dole ekranu. Po kliknięciu tego przycisku wyświetla się podsumowanie z opcją potwierdzenia lub anulowania zamówienia lub rezerwacji (Rysunek 103).



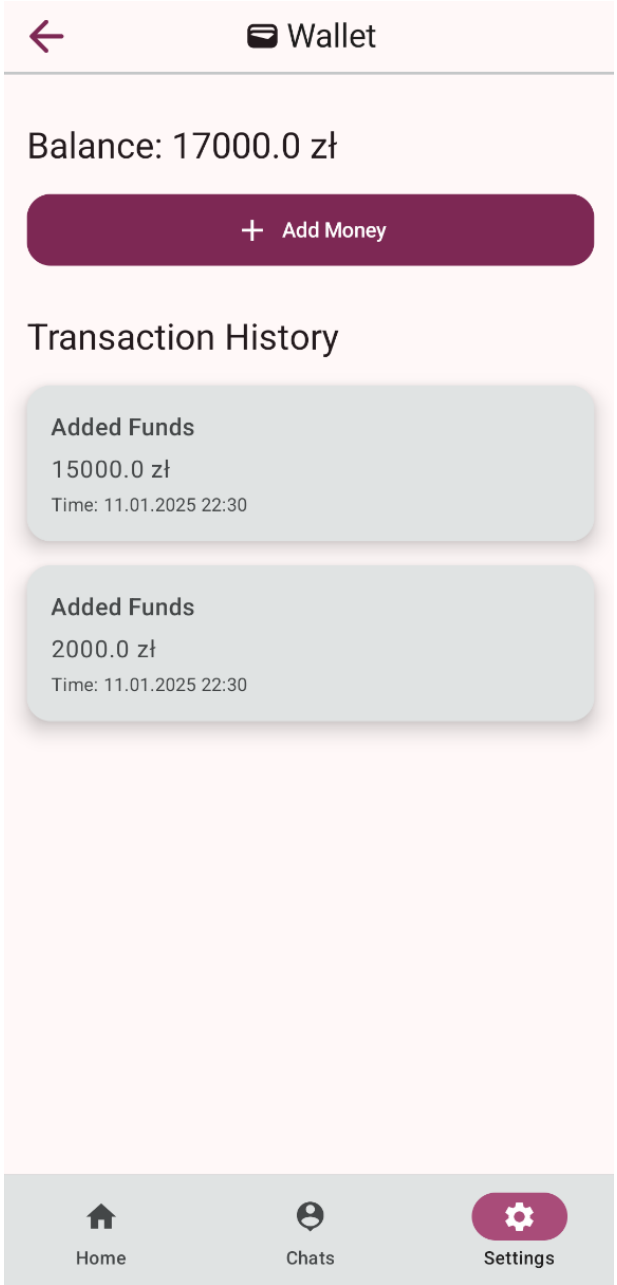
Rysunek 103 Podsumowanie rezerwacji. Źródło: Opracowanie własne

6.2.6 Ustawienia aplikacji i panel użytkownika

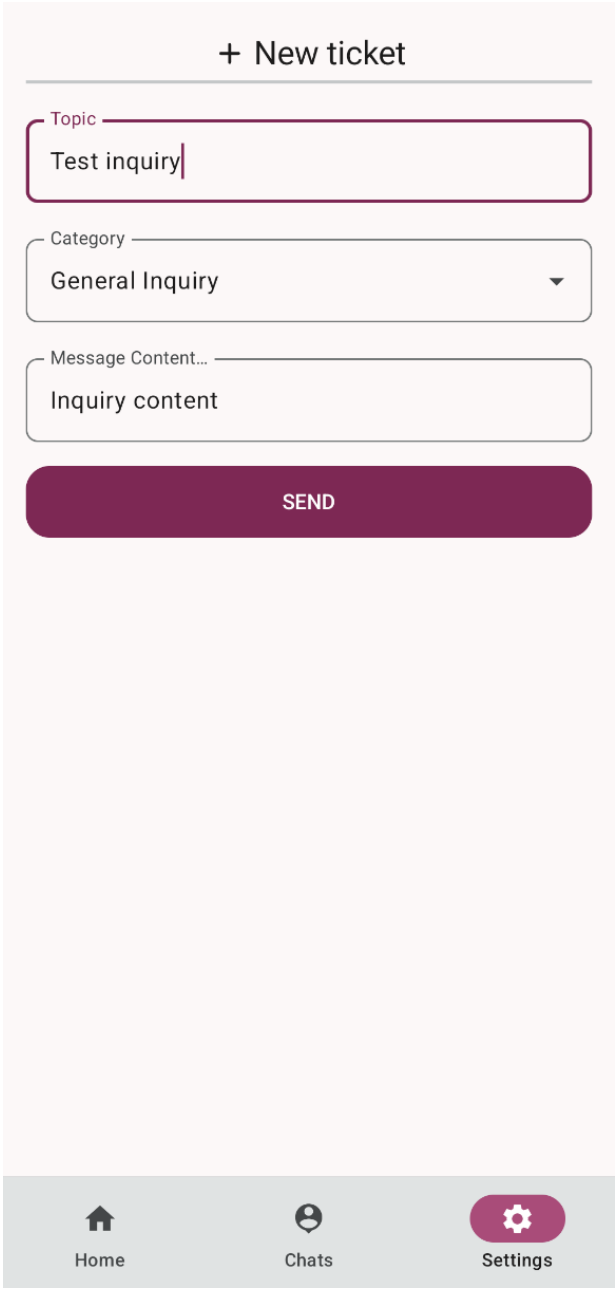
Po przejściu do zakładki ustawień użytkownikowi wyświetla się lista następujących zakładek:

1. Profil użytkownika
2. Portfel - użytkownik może dodać fundusze do portfela poprzez przycisk "*Add money*". Wpłacone środki można wykorzystać opłacając z góry składane w restauracji zamówienie lub wymaganą kaucję.
3. Moje zamówienia- w niniejszej zakładce wyświetlane są zamówienia złożone przez Klienta.
4. Skargi- wybierając wspomnianą zakładkę użytkownik może zobaczyć skargi, które złożył oraz skargi, które zostały na niego zgłoszone.

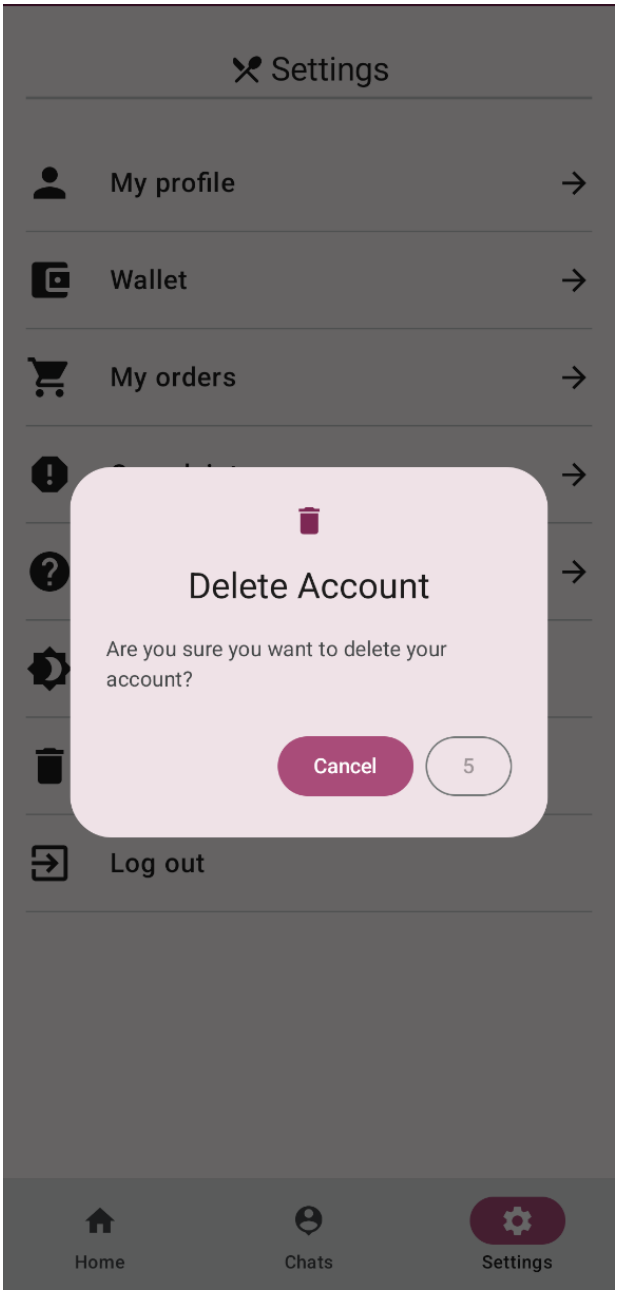
5. Helpdesk - służy do zgłaszania problemów do pracowników BOKu. Zgłoszenia mogą dotyczyć: wsparcia technicznego, rozliczeń lub być ogólnym zapytaniem.
6. Możliwość zmiany motywu z jasnego na ciemny
7. Opcja wylogowania
8. Opcja usunięcia swojego konta- przed zatwierdzeniem operacji usunięcia konta należy odczekać 5 sekund, po tym czasie pojawi się przycisk "*Yes*". Wdrożono wspomnianą funkcjonalność w celu uniknięcia przypadkowego usunięcia konta użytkownika.



Rysunek 104 Zakładka Portfel. Źródło: Opracowanie własne



Rysunek 105 Zakładka Helpdesk. Źródło: Opracowanie własne



Rysunek 106 Usunięcie konta użytkownika. Źródło: Opracowanie własne

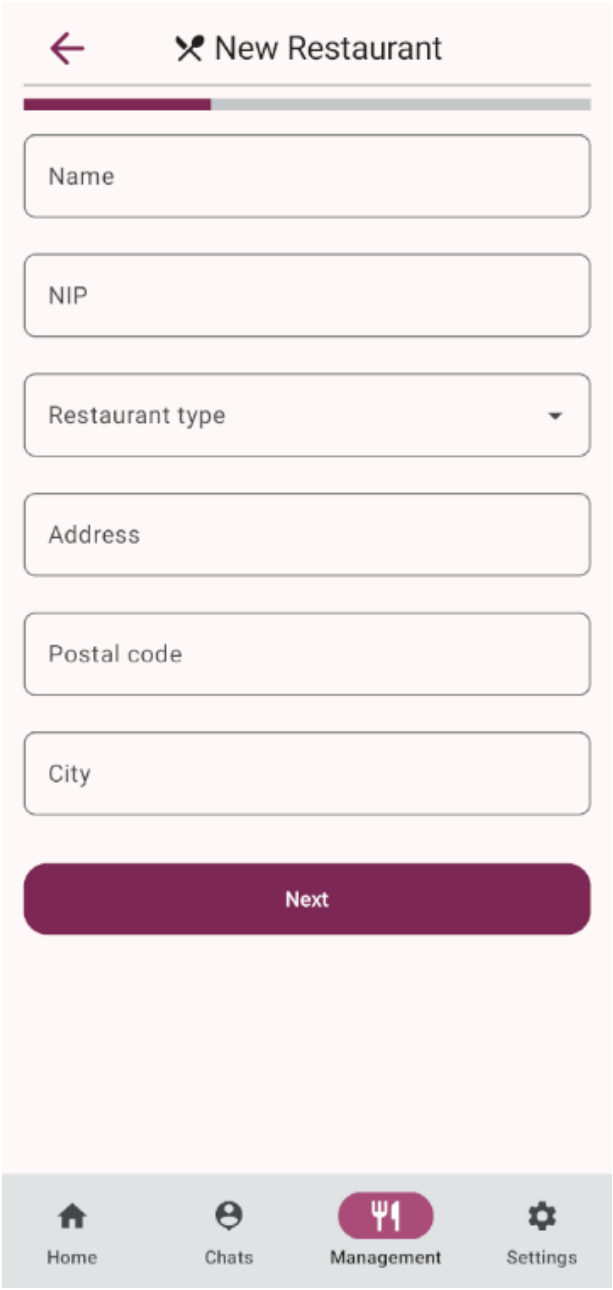
6.2.7 Widok właściciela restauracji

Logując się do aplikacji jako właściciel restauracji, dla użytkownika dostępna jest dodatkowo zakładka "*Management*".

Po otwarciu wspomnianej zakładki, użytkownik widzi wszystkie swoje zarejestrowane w systemie restauracje, tak jak zostało to pokazane poprzez Rysunek 107. Restauracje przypisane są do grup, co ułatwia właścicielowi wielu lokali zarządzanie nimi np. poprzez wyświetlenie statystyk dla konkretnej grupy restauracji. Można tego dokonać za pomocą przycisku "*See group stats*". Nazwę grupy można edytować za pomocą przycisku "*Edit group name*". Grupę, do której przynależy restauracja określa się przy dodawaniu restauracji do systemu.

Rysunek 107 Ekran zarządzania restauracjami. Źródło: Opracowanie własne

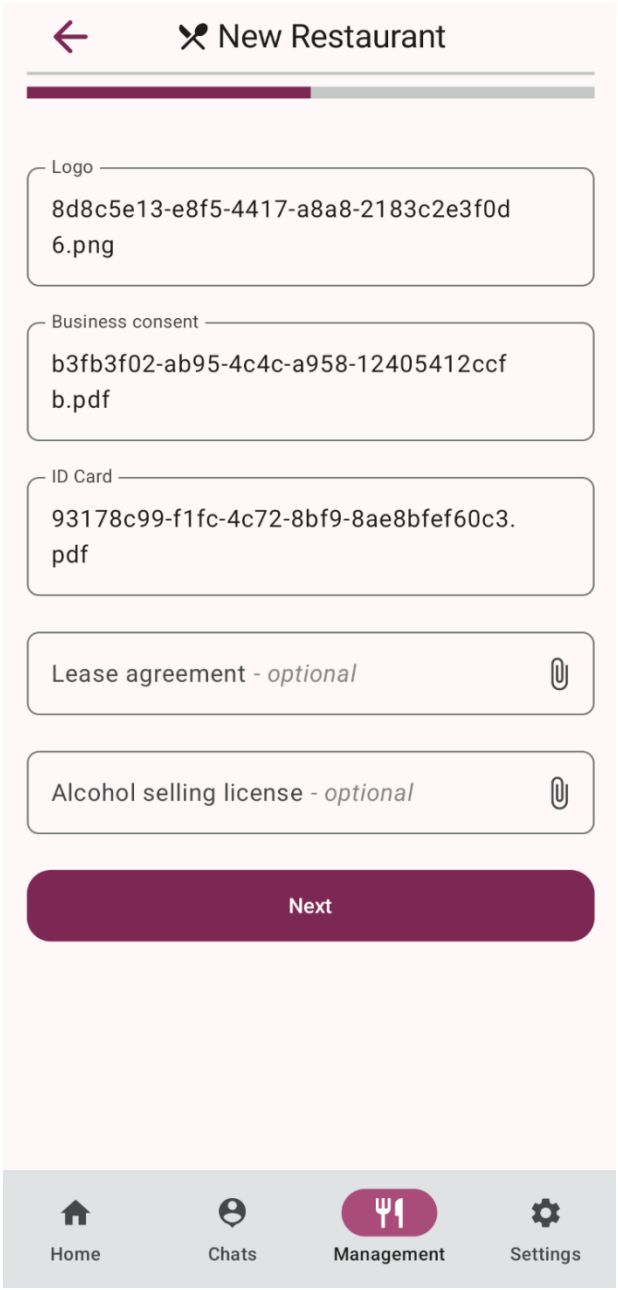
W prawym dolnym rogu znajduje się przycisk “+” pozwalający użytkownikowi na dodanie nowej restauracji. Po kliknięciu tego przycisku, użytkownikowi wyświetla się formularz danych do wypełnienia, dla nowej restauracji (Rysunek 108).



Rysunek 108 Dodanie nowej restauracji. Źródło: Opracowanie własne

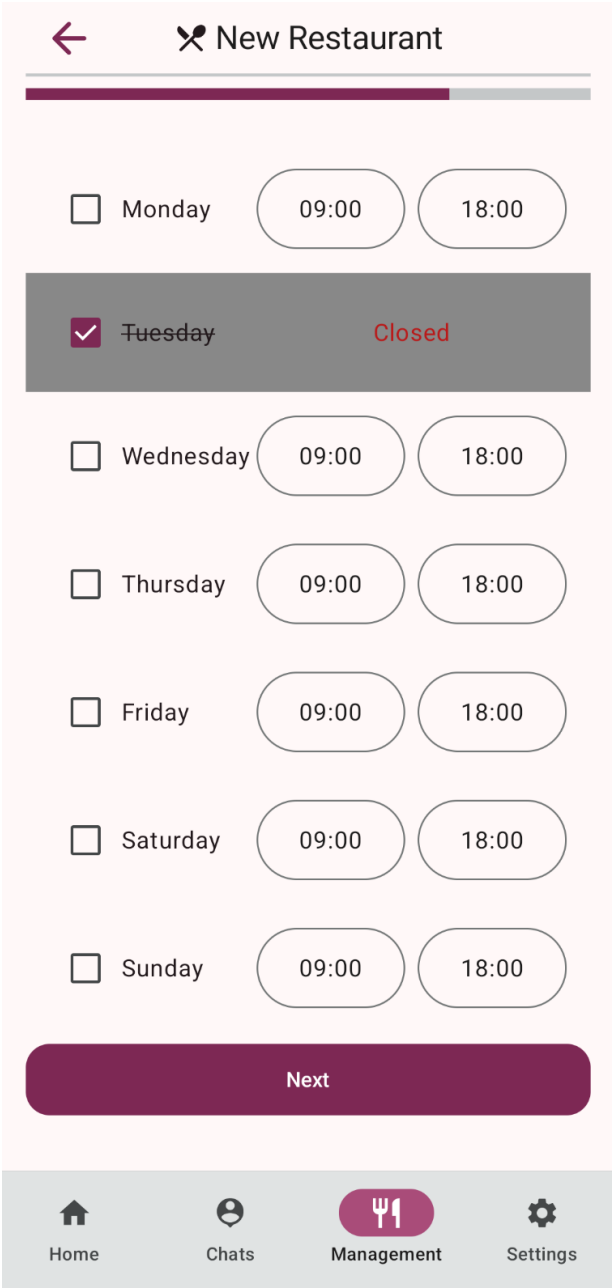
Po wprowadzeniu poprawnych danych lokalu, system prosi o dołączenie następujących dokumentów w celu poświadczenia własności (Rysunek 109):

1. Logo i galeria zdjęć (opcjonalnie,)
2. Pozwolenie na prowadzenie działalności,
3. Skan dokumentu tożsamości,
4. Umowa najmu (opcjonalnie),
5. Pozwolenie na sprzedaż alkoholu (opcjonalnie).



Rysunek 109 Etap drugi rejestracji restauracji. Źródło: Opracowanie własne.

Kolejnym krokiem jest określenie dni tygodnia oraz godzin otwarcia restauracji, jak pokazuje Rysunek 110.

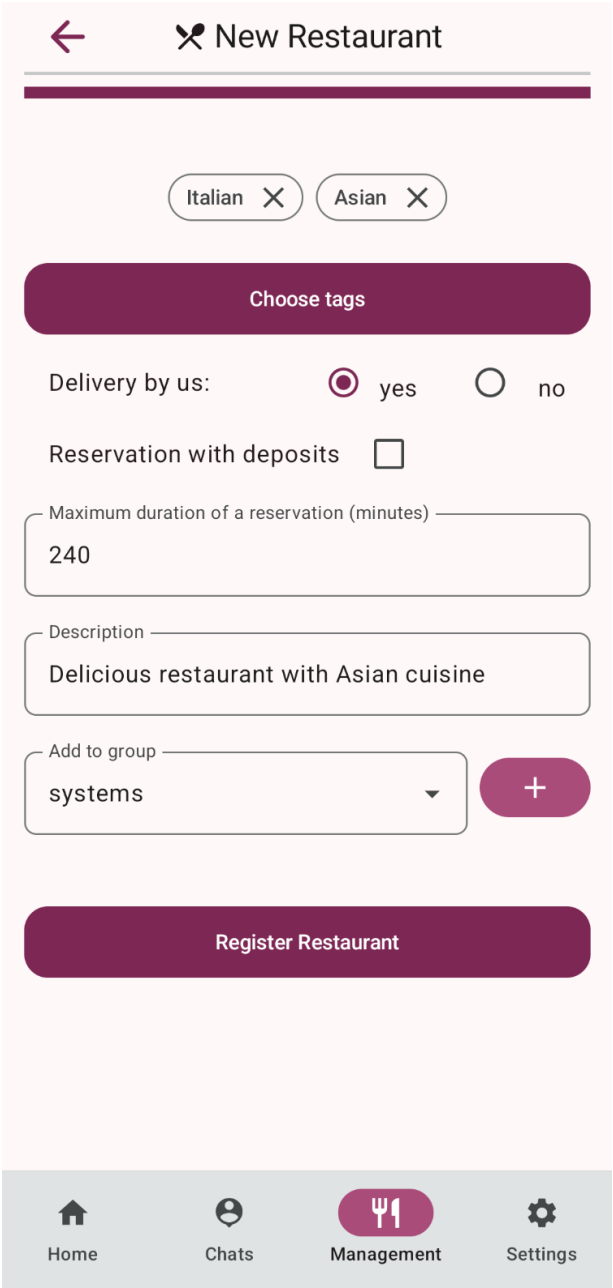


Rysunek 110 Etap trzeci rejestracji restauracji. Źródło: Opracowanie własne.

Końcowym etapem rejestracji restauracji (Rysunek 111) jest:

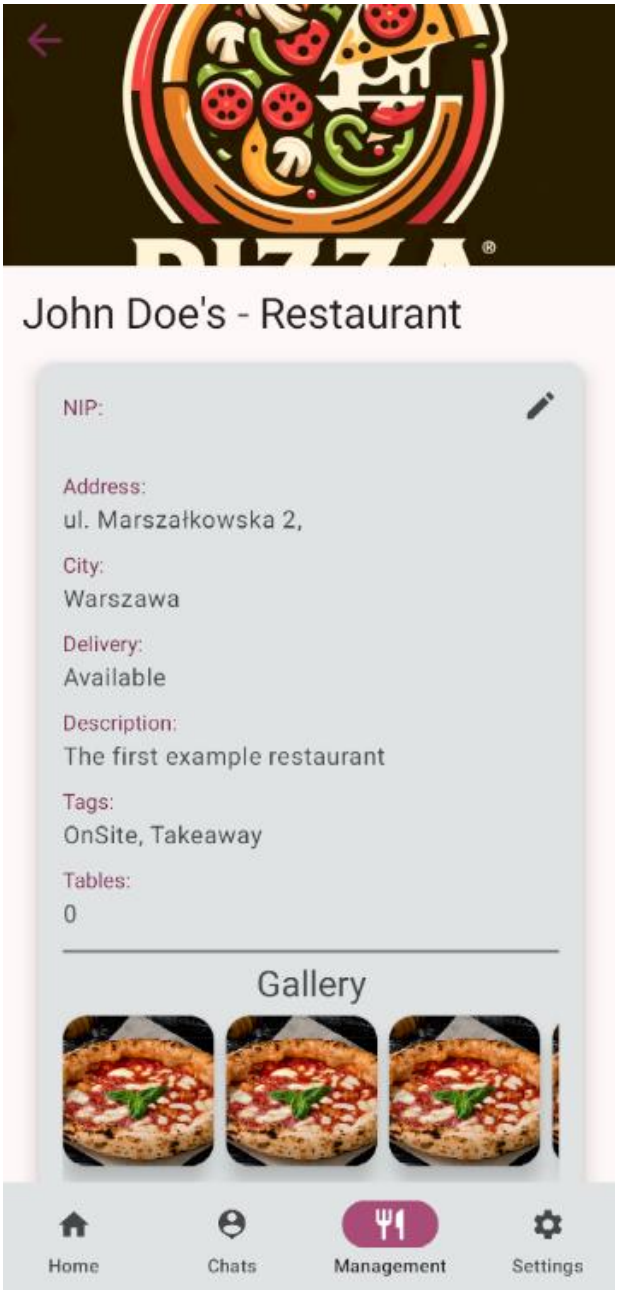
1. dodanie tagów, po których Klienci będą mogli wyszukiwać restaurację,
2. określenie czy lokal realizuje dostawy,
3. określenie czy rezerwacja wymaga depozytu,
4. określenie maksymalnego czasu rezerwacji stolika,
5. dodanie opisu,
6. dodanie restauracji do grupy restauracji.

Restauracje można dodać do istniejącej grupy restauracji wybierając ją z listy lub utworzyć nową grupę za pomocą ikony "+" znajdującej się obok wyboru grupy. Po wypełnieniu wszystkich pól i w celu rejestracji restauracji należy kliknąć przycisk "*Register Restaurant*".



Rysunek 111 Ostatni etap rejestracji restauracji. Źródło: Opracowanie własne

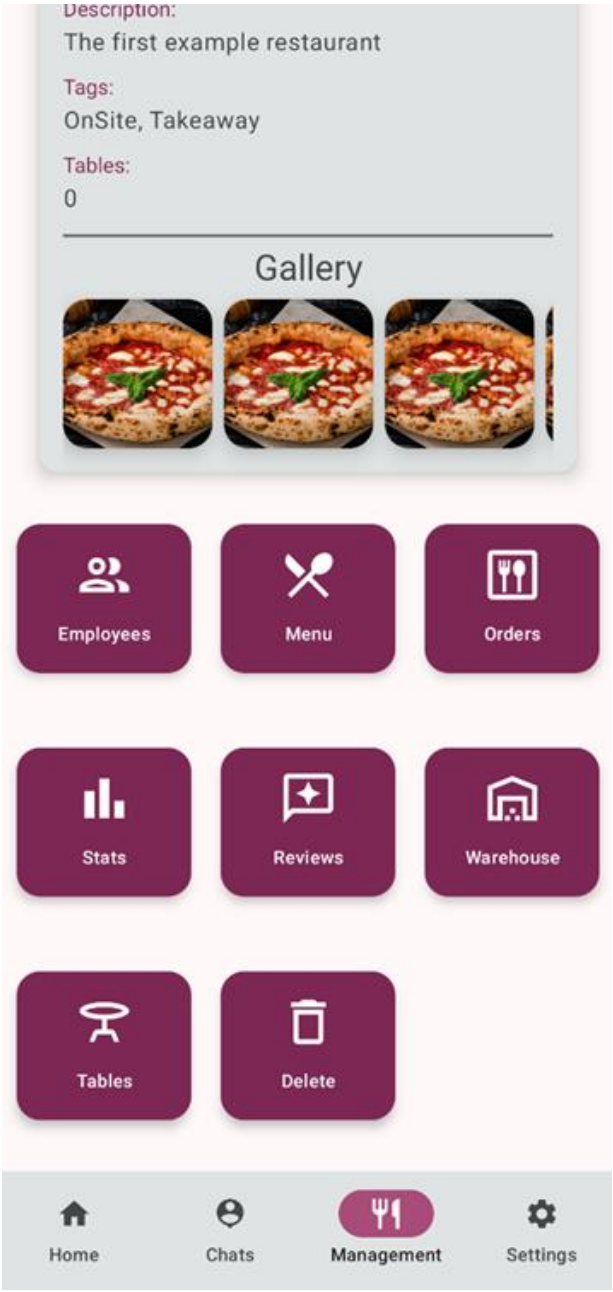
Po kliknięciu w daną restaurację (wychodząc od ekranu głównego "*Management*"), użytkownikowi wyświetla się ekran z danymi oraz ze zdjęciami z wybranej restauracji (Rysunek 112). Użytkownik ma możliwość edycji tych danych po kliknięciu ikony ołówka znajdującej się w prawym górnym rogu ekranu.



Rysunek 112 Ekran informacji o restauracji. Źródło: Opracowanie własne

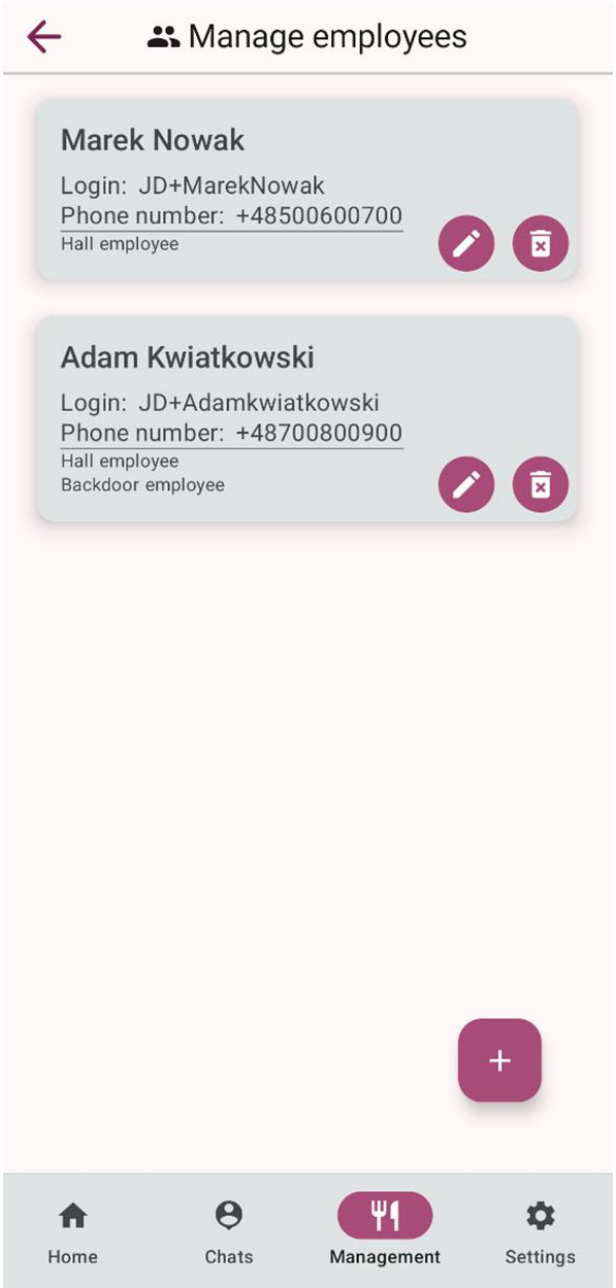
Po przewinięciu do dołu, użytkownik widzi przyciski przekierowujące do zarządzania konkretnymi obszarami dla wybranej restauracji (Rysunek 113). Użytkownik może zarządzać:

1. pracownikami,
2. menu lokalu,
3. zamówieniami,
4. opiniami Klientów o lokalu,
5. dostawami produktów do lokalu,
6. stolikami,
7. oraz ma możliwość usunięcia restauracji.



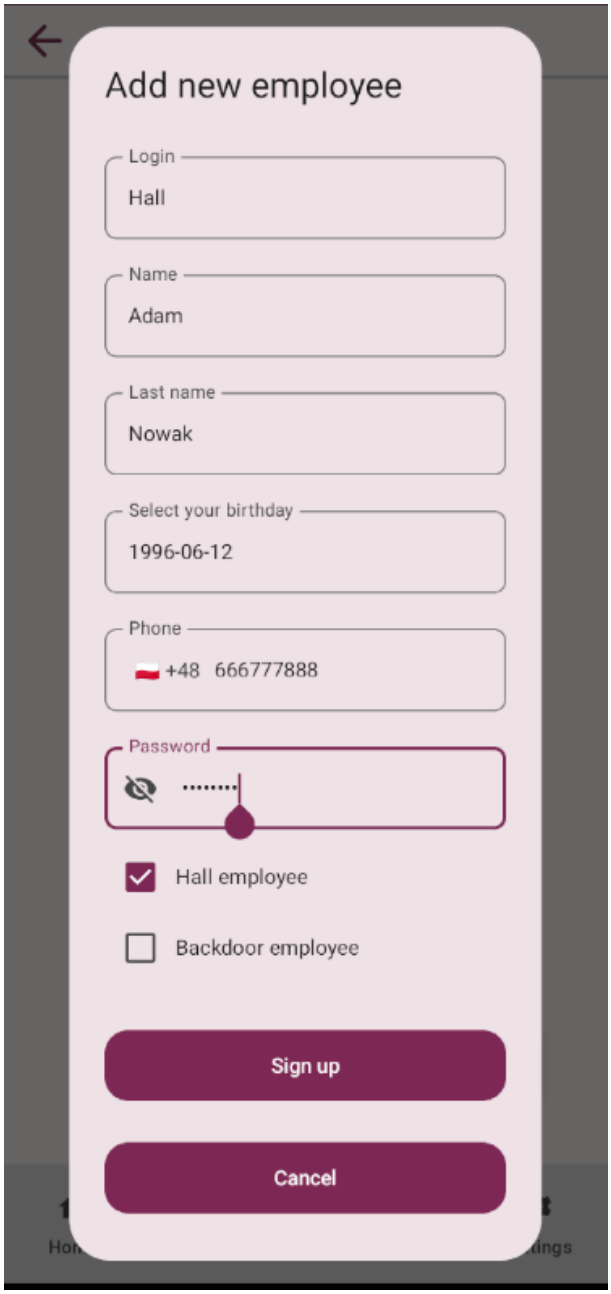
Rysunek 113 Panel zarządzania restauracją. Źródło: Opracowanie własne

Po przejściu do ekranu zarządzania pracownikami wyświetlają się pracownicy zarejestrowani w danej restauracji (Rysunek 114). Użytkownik zalogowany jako właściciel restauracji może edytować istniejących pracowników i dodawać nowych za pomocą plusa znajdującego się w prawym dolnym rogu ekranu.



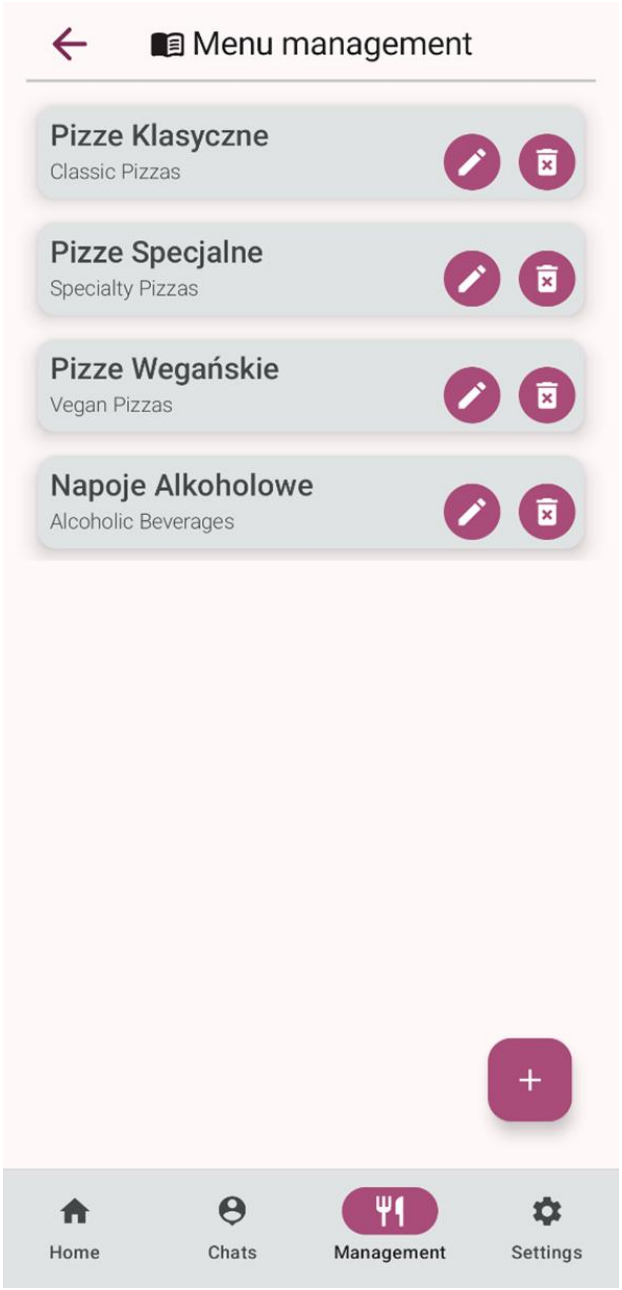
Rysunek 114 Zarządzanie pracownikami. Źródło: Opracowanie własne

Po kliknięciu “+”, wyświetla się formularz, który należy wypełnić danymi w celu dodania nowego pracownika restauracji (Rysunek 115).

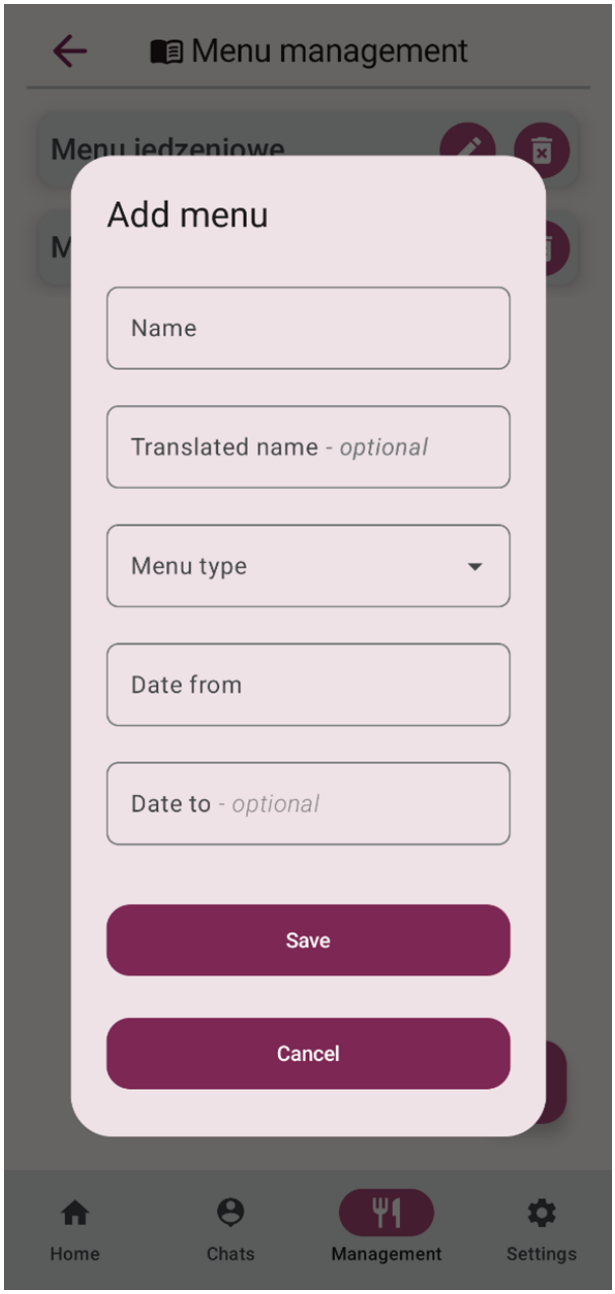


Rysunek 115 Formularz dodania nowego pracownika. Źródło: Opracowanie własne

Po przejściu do Menu, z ekranu zarządzania restauracją wyświetlają się menu zarejestrowane dla danej restauracji. Menu można dodawać oraz edytować istniejące, tak jak ilustruje to Rysunek 116 oraz Rysunek 117.

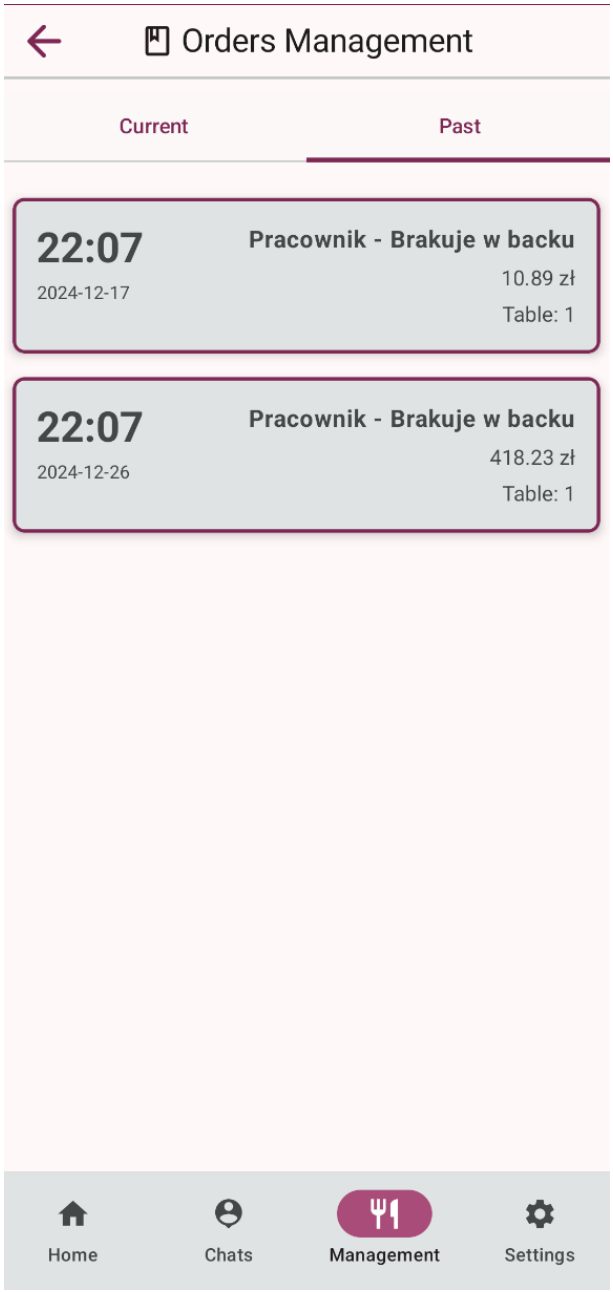


Rysunek 116 Ekran zarządzania menu restauracji. Źródło: Opracowanie własne



Rysunek 117 Formularz dodania nowego menu. Źródło: Opracowanie własne

W zakładce "*Orders*" (Rysunek 118) właściciel restauracji może przeglądać obecne oraz minione zamówienia z lokalu, przełączając zakładki "*Current*" oraz "*Past*" znajdujące się na górze ekranu.



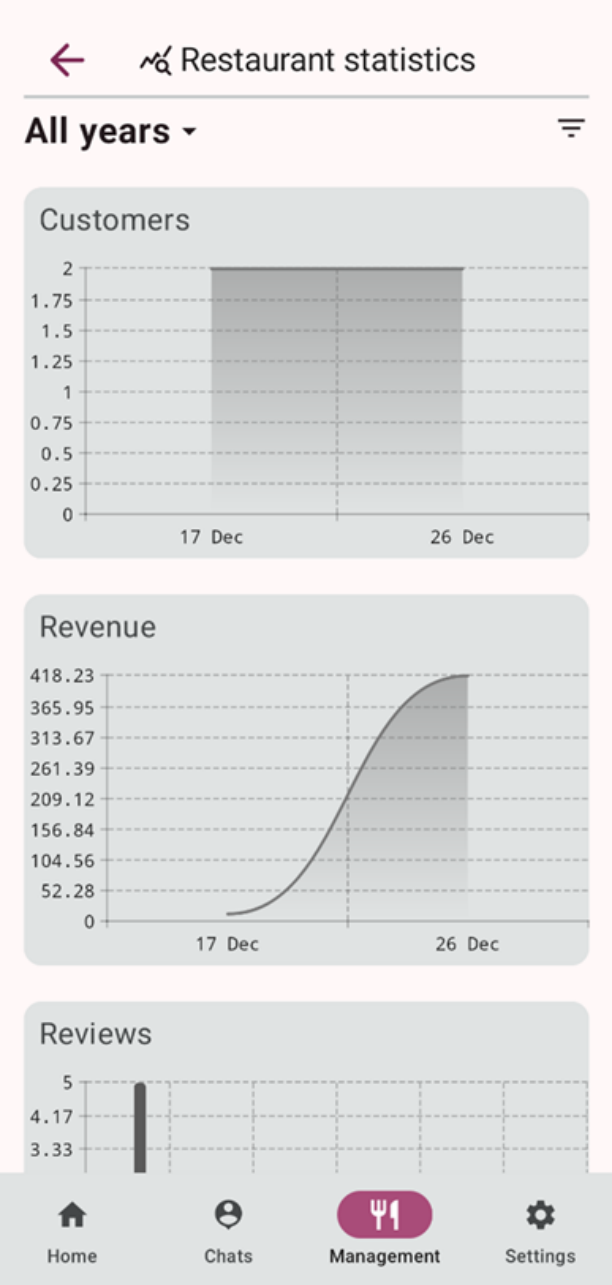
Rysunek 118 Ekran zarządzania zamówieniami. Źródło: Opracowanie własne

Do aplikacji może zalogować się, także pracownik restauracji. Dla pracownika pracującego w więcej niż jednej restauracji, po zalogowaniu się widoczne są wszystkie restauracje, w których pracuje oraz wiadomość powitalną.

W zakładce "*Stats*", w którą można wejść z poziomu ekranu głównego zarządzania restauracją, wyświetlane są statystyki lokalu, tak jak pokazuje to Rysunek 119. Na ekranie prezentowane są cztery wykresy dotyczące odpowiednio:

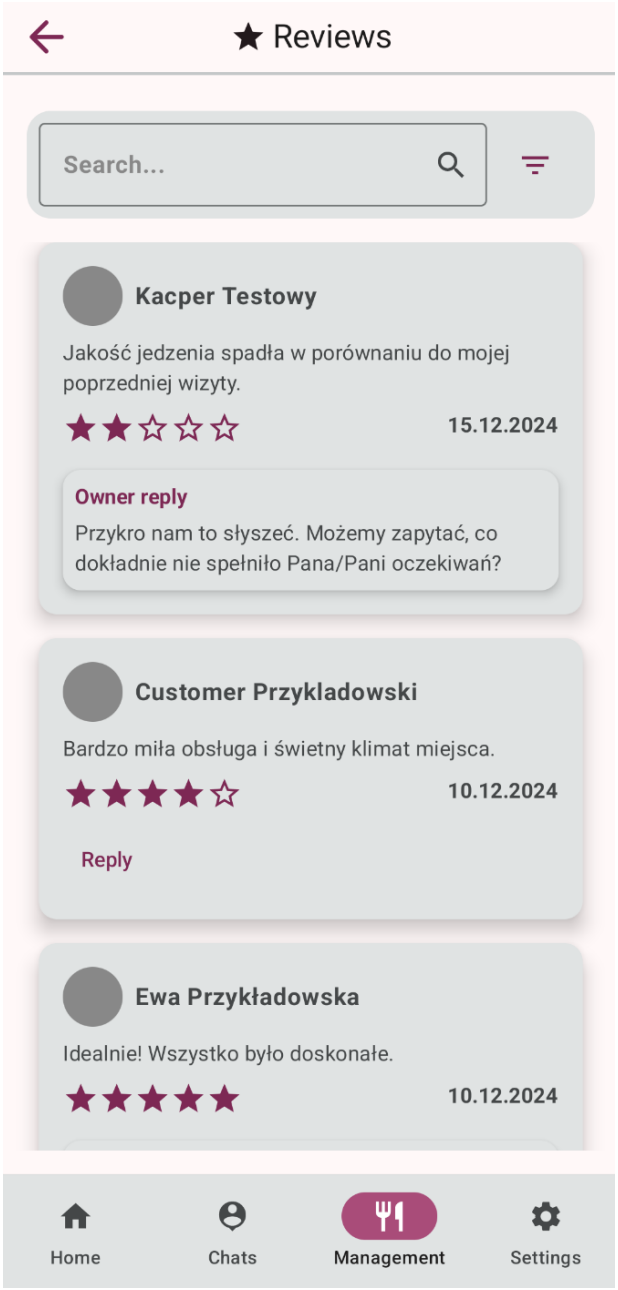
1. liczby klientów w lokalu,
2. wysokości dochodu lokalu w zadanym okresie,
3. liczby wystawionych opinii,
4. średni dochód lokalu.

Statystyki przedstawiają wszystkie dane zebrane na przestrzeni lat lub dane z zadanego okresu, użytkownik wybiera rok z listy znajdującej się w lewym górnym rogu ekranu a następnie określa miesiąc.



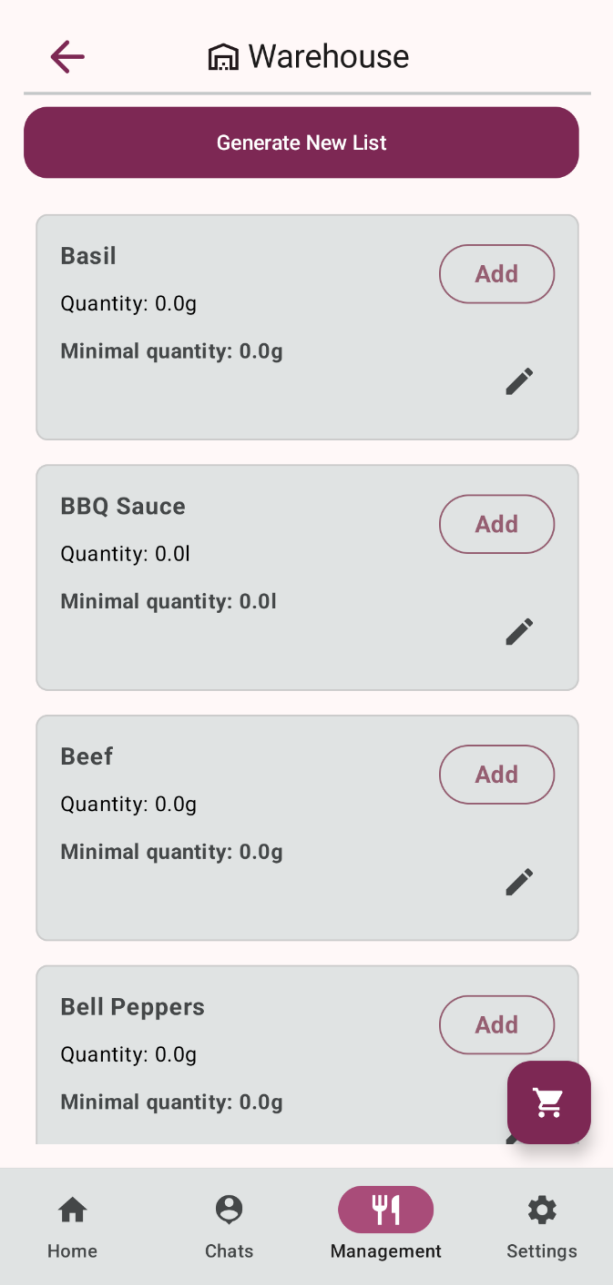
Rysunek 119 Ekran wykresów i statystyk. Źródło: Opracowanie własne

Po przejściu do zakładki "*Reviews*", właściciel lokalu może przeglądać i odpowiadać na opinie Klientów. Opinie można wyszukiwać wpisując oczekiwany tekst w wyszukiwarkę oraz filtrować na podstawie liczby wystawionych gwiazdek. Aby odpowiedzieć na komentarz Klienta, należy kliknąć przycisk "*Reply*" znajdujący się pod wybranym komentarzem. Ekran opini pokazuje Rysunek 120.

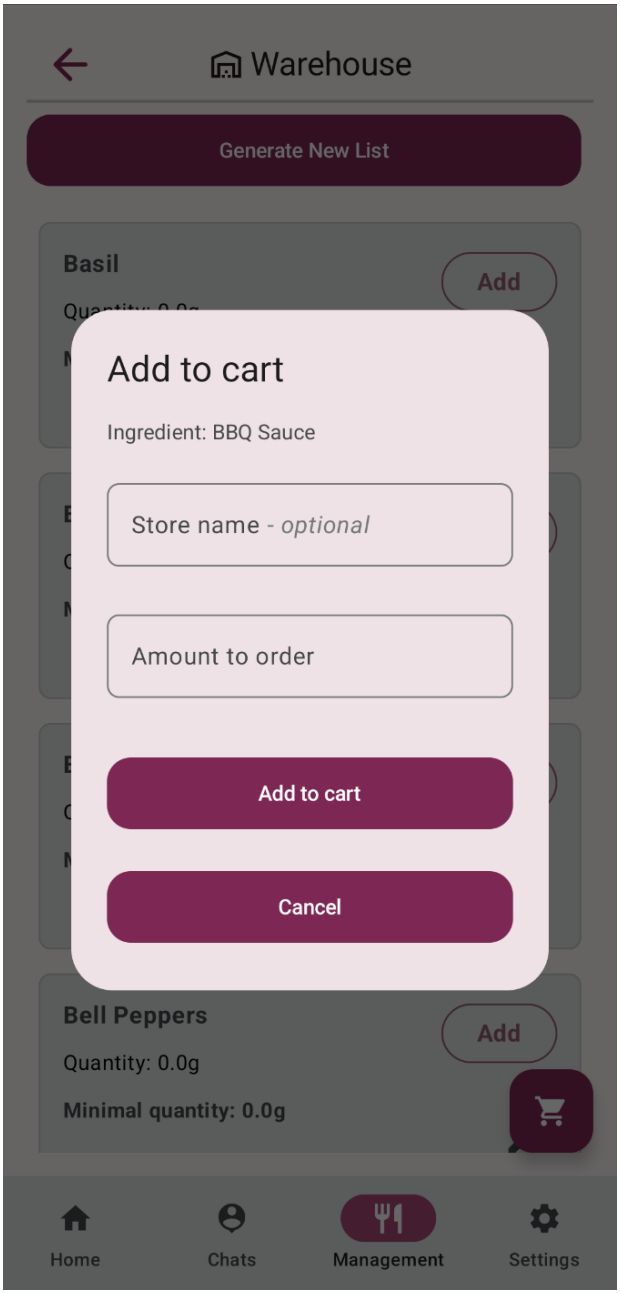


Rysunek 120 Zakładka odpowiedzi na opinie

W zakładce "*Warehouse*" właściciel restauracji może zarządzać dostawami do lokalu. Przycisk znajdujący się na górze ekranu "*Generate New List*" odpowiada za dodanie wszystkich brakujących w restauracji, regularnie zamawianych, składników do koszyka. Do koszyka można dodać też inne składniki z listy poprzez kliknięcie przycisku "*Add*" przy wybranym składniku. Ekran dostaw został pokazany przez Rysunek 121. Zamawiający podaje liczbę towaru, który chce zamówić oraz może opcjonalnie określić nazwę sklepu, z którego chce zamówić składnik, jak prezentuje Rysunek 122.



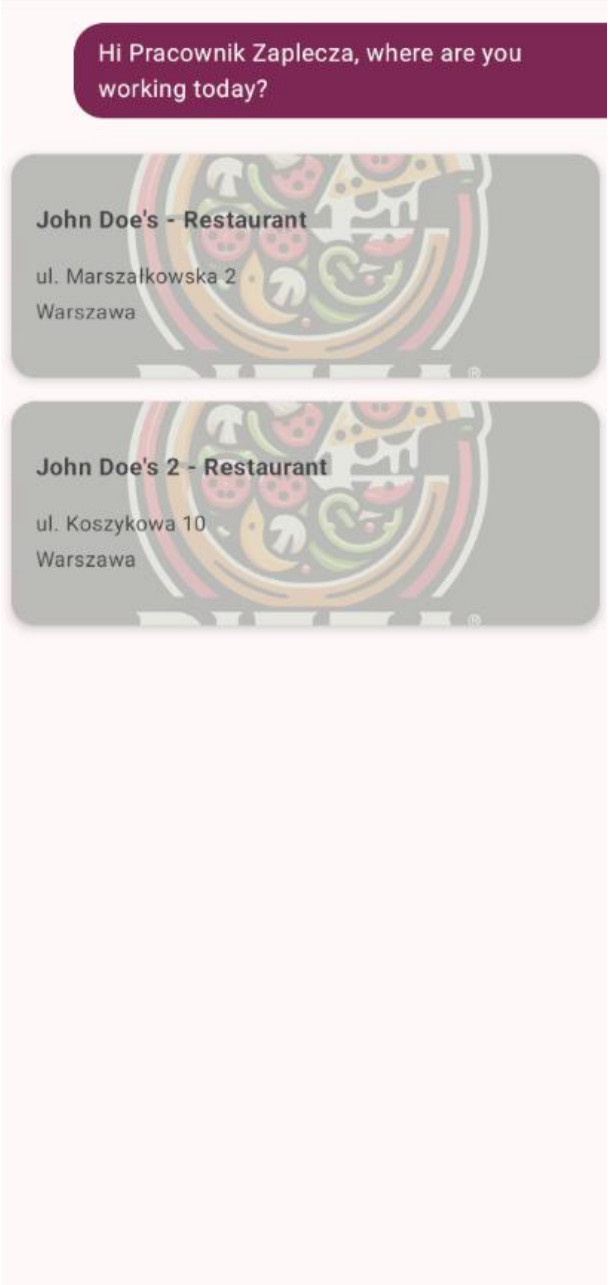
Rysunek 121 Ekran zarządzania asortymentem. Źródło: Opracowanie własne



Rysunek 122 Ekran dodania składnika do zamówienia. Źródło: Opracowanie własne.

6.2.8 Widok pracownika restauracji

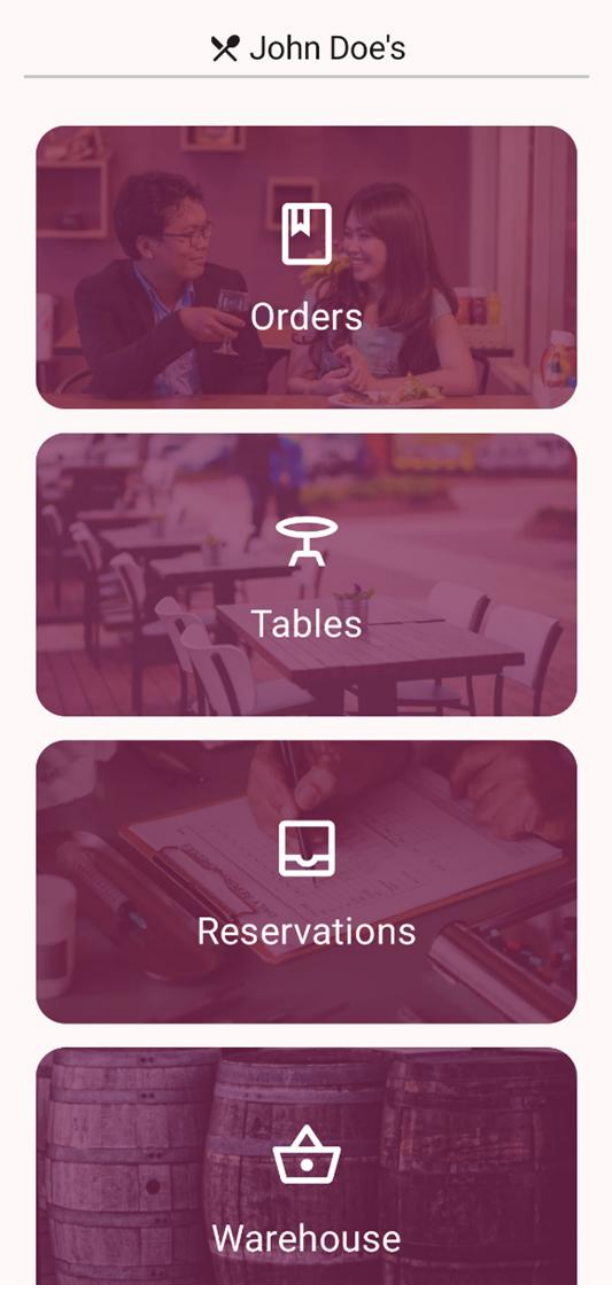
Do aplikacji może zalogować się, także pracownik restauracji. Dla pracownika pracującego w więcej niż jednej restauracji, po zalogowaniu się widoczne są wszystkie restauracje, w których pracuje oraz wiadomość powitalna (Rysunek 123).



Rysunek 123 Ekran wyboru restauracji przez pracownika. Źródło: Opracowanie własne

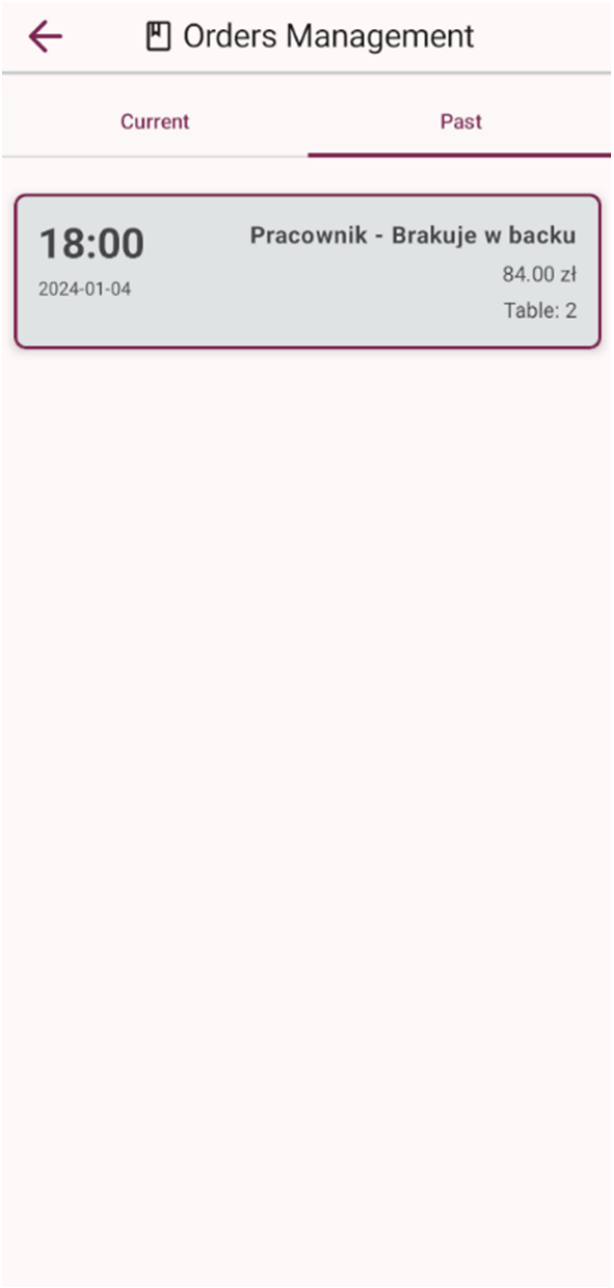
Po wybraniu restauracji przez pracownika bądź dla pracowników pracujących tylko w jednej restauracji, widoczny jest ekran zilustrowany przez Rysunek 124. Na górze ekranu wyświetlany jest nagłówek z nazwą restauracji, poniżej dostępne są następujące zakładki:

1. "*Orders*" - zakładka z informacjami o zamówieniach
2. "*Tables*" - w zakładce znajdują się informacje dotyczące dostępności stolików
3. "*Reservations*" - z informacjami o rezerwacjach
4. "*Warehouse*" - służąca do zarządzania zaopatrzeniem lokalu
5. "*Settings*" - zakładka, w której znajdują się ustawienia



Rysunek 124 Panel pracownika restauracji. Źródło: Opracowanie własne

Po otwarciu zakładki "*Orders*", służącej do zarządzania zamówieniami, pracownikowi wyświetla się ekran z przyjętymi, niezrealizowanymi jeszcze zamówieniami - "*Current*". Po zmianie zakładki na "*Past*" pracownik może przeglądać przeszłe zamówienia (Rysunek 125).

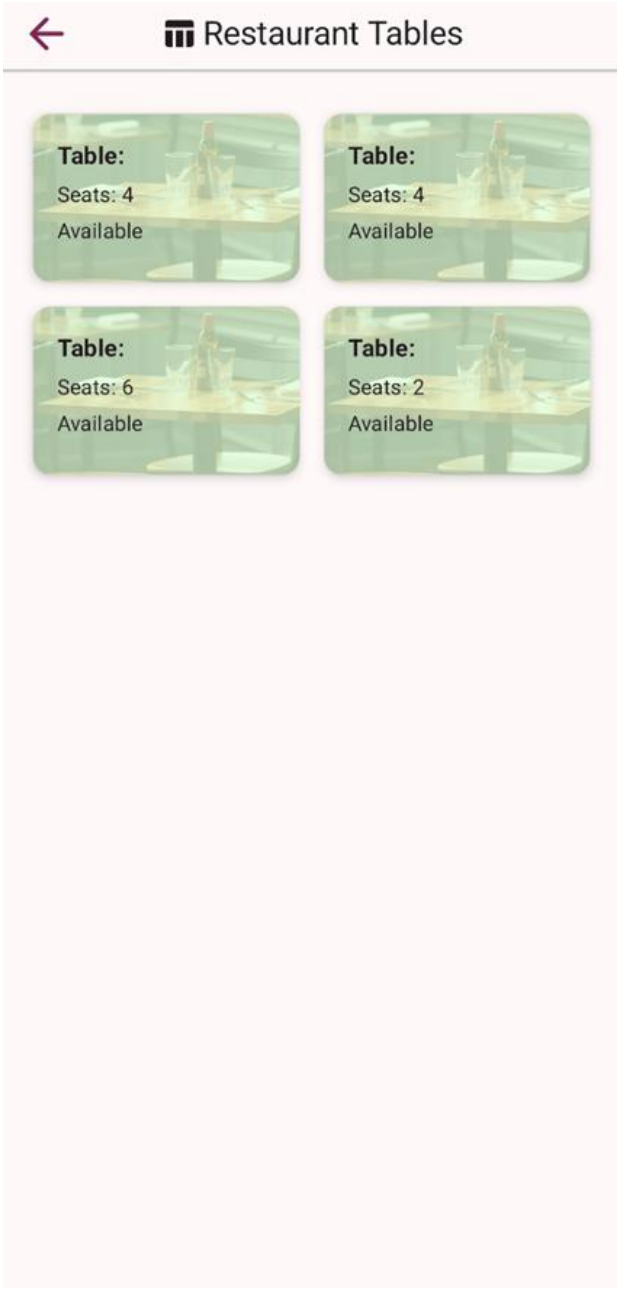


Rysunek 125 Ekran zarządzania zamówieniami. Źródło: Opracowanie własne

Zakładka "*Tables*" (Rysunek 126) służy do zarządzania stolikami. Pracownik może sprawdzić obecny status stolików w restauracji. Wyróżniamy następujące statusy dla stolików:

1. Dostępny,
2. Zarezerwowany,
3. Zajęty.

Status stolika zmienia się automatycznie ze statusu "Dostępny" na "Zarezerwowany", gdy stolik zostaje przypisany do rezerwacji. Kiedy pracownik restauracji potwierdzi rezerwację w aplikacji, status stolika zostanie automatycznie zmieniony na "Zajęty"

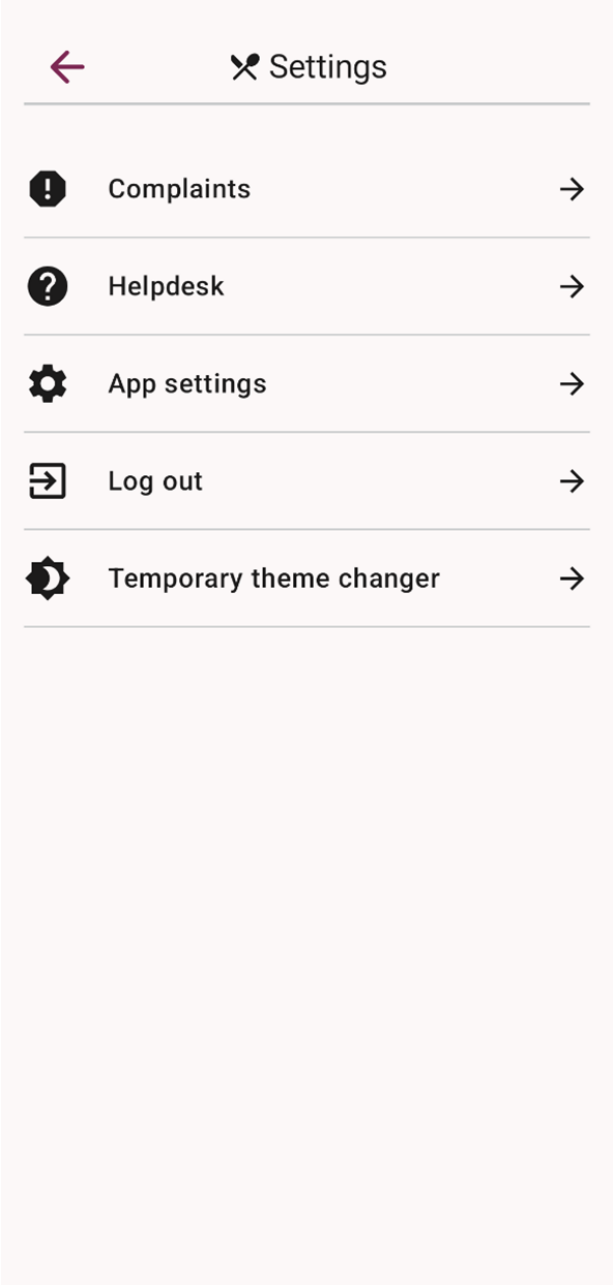


Rysunek 126 Ekran zarządzania stolikami. Źródło: Opracowanie własne

Po przejściu do zakładki "*Reservations*" (Rysunek 127) pracownik może zobaczyć aktualne rezerwacje złożone przez klientów restauracji.

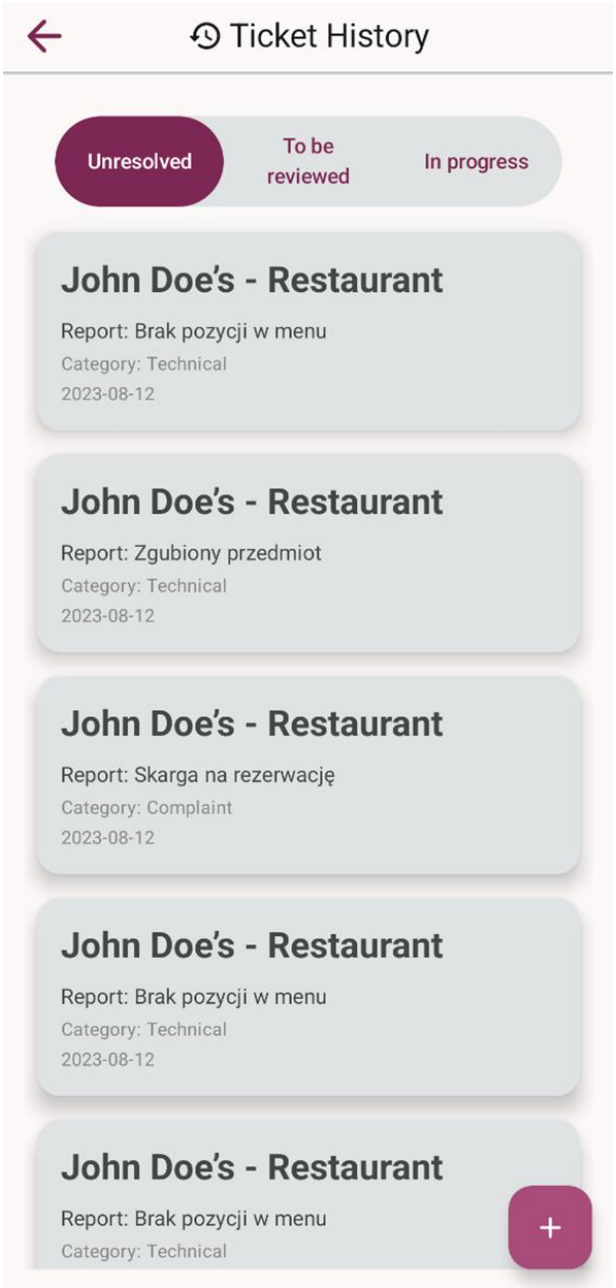
Rysunek 127 Ekran zarządzania rezerwacjami. Źródło: Opracowanie własne

Dla pracowników, jak i dla wszystkich użytkowników aplikacji, dostępna jest zakładka "*Settings*" (Rysunek 128), w której dostępne są podstawowe opcje ustawień oraz dodatkowe funkcjonalności.



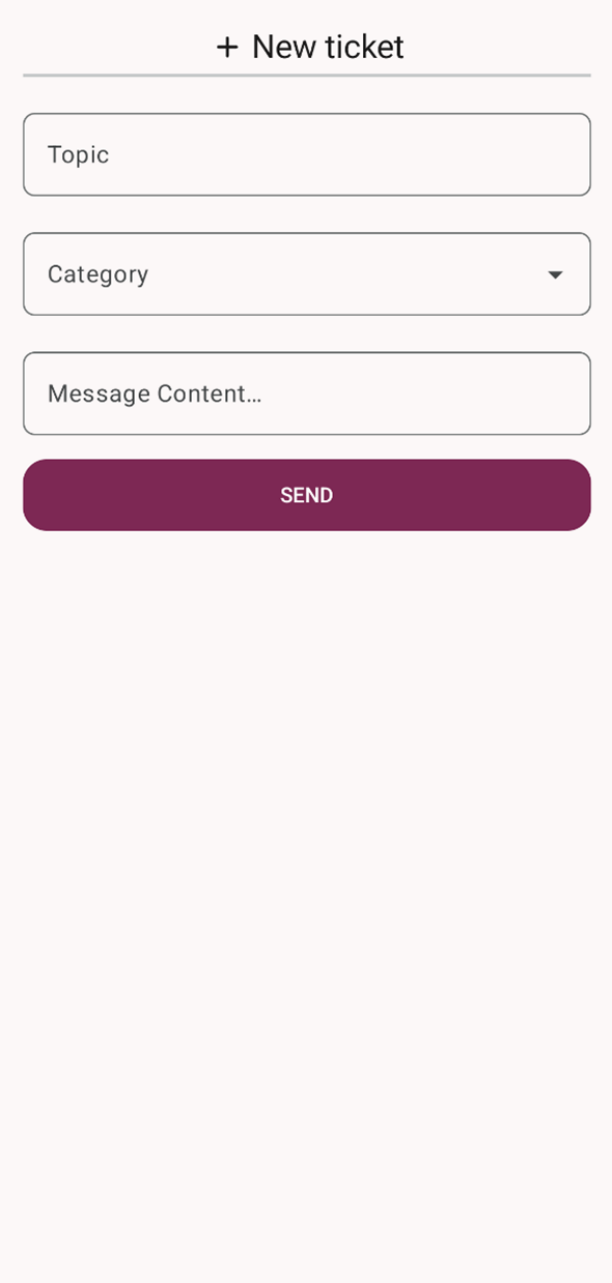
Rysunek 128 Ekran ustawień aplikacji mobilnej. Źródło: Opracowanie własne

Po przejściu do "*Complaints*" (Rysunek 129) widoczne są skargi. Skargi mogą być w różnym statusie rozpatrywania i zależnie od tego statusu będą wyświetlane na stronie "*Unresolved*" - nierozwiązane, "*To be reviewed*" - do weryfikacji, "*In progress*" - w trakcie.



Rysunek 129 Panel złożonych skarg. Źródło: Opracowanie własne

Za pomocą plusa w prawym dolnym rogu można dodać nową skargę, w tym celu po kliknięciu wyświetli się formularz, który przedstawia Rysunek 130.



Rysunek 130 Formularz dodania nowego zgłoszenia lub skargi. Źródło: Opracowanie własne

W zakładce "*Helpdesk*" tworzy się zgłoszenia do pracowników BOK. Zgłoszenia mogą być zarówno ogólnym zapytaniem, dotyczyć wsparcia technicznego czy rozliczeń płatności. Rodzaj zgłoszenia określa się w zakładce "*Category*".

7 Testowanie aplikacji

Testowanie oprogramowania jest kluczowym etapem w cyklu życia projektu, który pozwala na zapewnienie wysokiej jakości produktu, wykrycie i naprawę błędów oraz potwierdzenie zgodności systemu z wymaganiami. W niniejszym rozdziale zostaną opisane procesy i narzędzia używane do testowania poszczególnych warstw systemu, w tym backendu, frontendu oraz aplikacji mobilnej. Każda z tych warstw odgrywa istotną rolę w zapewnieniu niezawodności, wydajności i użyteczności systemu.

Każda sekcja zawiera szczegółowe opisy stosowanych narzędzi, procesów testowych oraz metryk, które posłużyły do oceny jakości systemu.

7.1 Testy aplikacji webowej

W projekcie do automatyzacji testów aplikacji webowej zostało wykorzystane Selenium, a same testy zostały napisane w języku Python. Testy zostały podzielone na moduły, z których każdy odpowiada za sprawdzenie konkretnej funkcjonalności aplikacji. Dzięki wykorzystaniu tzw. webdriverów, Selenium umożliwiło zautomatyzowanie interakcji z aplikacją, co pozwoliło na efektywne przeprowadzanie testów, takich jak symulowanie działań użytkownika, wypełnianie formularzy, klikanie przycisków czy sprawdzanie reakcji aplikacji na różne scenariusze.

Do przykładów należą:

- Test rejestracji użytkownika, gdzie sprawdzane są pola formularza, walidacja danych i zmiana stron,
- Test rejestracji restauracji, który dodatkowo obsługuje funkcję przesyłania plików takich jak logo, umowy czy licencje na alkohol,
- Test akceptowania zaproszeń znajomych, gdzie symulowane są interakcje z funkcjami powiadomień.

Zostały opracowane funkcje wspomagające, które rozszerzają możliwości Selenium i dostosowują je do potrzeb testowania aplikacji. Każda funkcja ma dwie główne cechy: oczekiwanie na pojawienie się elementu na stronie oraz parametr critical, który pozwala na przerwanie testu lub kontynuowanie po błędzie.

Przykładem takich funkcji są m.in. `wait_for_element` (Listing 70) oraz `click_button` (Listing 71), które pomagają w automatyzacji działań na stronie i obsłudze błędów.

Listing 70 Przykład funkcji wait_for_element. Źródło: Opracowanie własne

```
    def wait_for_element(driver, selector_type, selector_value, critical=True):
        """
        Czeka na obecność elementu na stronie w określonym czasie.
        Jeśli critical=True, rzuca wyjątek przy niepowodzeniu. Jeśli False, tylko loguje
błąd.
        """
        try:
            element = WebDriverWait(driver, timeout).until(
                EC.presence_of_element_located((selector_type, selector_value))
            )
            result(f"Element with {selector_type} '{selector_value}' is present on the
page.")
            return element
        except TimeoutException as e:
            result(
                f"Element with {selector_type} '{selector_value}' is not present on the
page within {timeout} seconds: {str(e)}",
                False)
```

```
            if critical:
                raise TimeoutException(
                    f"Element with {selector_type} '{selector_value}' is not present on the
page within {timeout} seconds.") from e
```

Listing 71 Przykład funkcji click_button. Źródło: Opracowanie własne

```
    def click_button(driver, selector_type, selector_value, critical=True):
        """
        Znajduje i klika przycisk na stronie używając podanego selektora.
        Czeka maksymalnie TIMEOUT sekund, aż przycisk stanie się dostępny.
        Jeśli critical=True, rzuca wyjątek przy niepowodzeniu. Jeśli False, tylko loguje
błąd.
        """
        try:
            # Oczekiwanie na element
            wait_for_element(driver, selector_type, selector_value, critical)

            # Znajdowanie elementu po oczekiwaniu
            button = driver.find_element(selector_type, selector_value)
            button.click()
            result(f"Successfully clicked button with {selector_type} '{selector_value}'")
        except (NoSuchElementException, TimeoutException) as e:
            result(f"Couldn't click button with {selector_type} '{selector_value}':
{str(e)}", False)
            if critical:
                raise Exception(f"Couldn't click button with {selector_type}
'{selector_value}'") from e
```

Do generowania danych testowych została wykorzystana klasa *RandomData*, która korzysta z biblioteki Faker i algorytmów do tworzenia danych osobowych, biznesowych czy związanych z menu. Te dane są używane do wypełniania formularzy w testach, np. przy rejestracji użytkownika.

Organizacja testów: Każdy test znajduje się w osobnym pliku, co ułatwia jego utrzymanie i dalszy rozwój kodu. Listing 72 prezentuje przykładowy test sprawdzający działanie rejestracji użytkownika w aplikacji webowej

Listing 72 Test rejestracji użytkownika, Źródło: Opracowanie własne

```
    def test_user_register(driver):
       info("USER REGISTER TEST")
       driver.get(register_url)
       try:
           # Sprawdzenie tytułu
           check_page_title(driver, "React App")


           # Sprawdzenie czy istnieje elemenet o id root, w naszym przypadku div
           wait_for_element(driver, By.ID, "root")


           # LOGIN
           # Znalezienie i kliknięcie przycisku
           click_text_field(driver, By.ID, "login")


           press_tab_key(driver)
```

```
            # Sprawdzenie czy pojawia się nowa zawartość
            wait_for_element(driver, By.ID, "login-helper-text")


            # EMAIL
            click_text_field(driver, By.ID, "email")


            press_tab_key(driver)


            wait_for_element(driver, By.ID, "email-helper-text")


            # PASSWORD
            click_text_field(driver, By.ID, "password")


            press_tab_key(driver)


            wait_for_element(driver, By.ID, "password-helper-text")


            # CONFIRM PASSWORD
            click_text_field(driver, By.ID, "confirmPassword")


            press_tab_key(driver)


            wait_for_element(driver, By.ID, "confirmPassword-helper-text")


            wait_for(delay)


            # Znaleznienie pola i wypełnienie go danymi


            enter_text(driver, By.ID, "login", RandomData.generate_login())


            enter_text(driver, By.ID, "email", RandomData.generate_email())


            password = RandomData.generate_password()


            enter_text(driver, By.ID, "password", password)


            enter_text(driver, By.ID, "confirmPassword", password)
            result("Form filled correctly")


            wait_for(delay)


            # Czekamy na pojawienie się przycisku
            universal_wait_for(driver, EC.element_to_be_clickable, By.CSS_SELECTOR,
"button")


            click_button(driver, By.CSS_SELECTOR, "button")
```

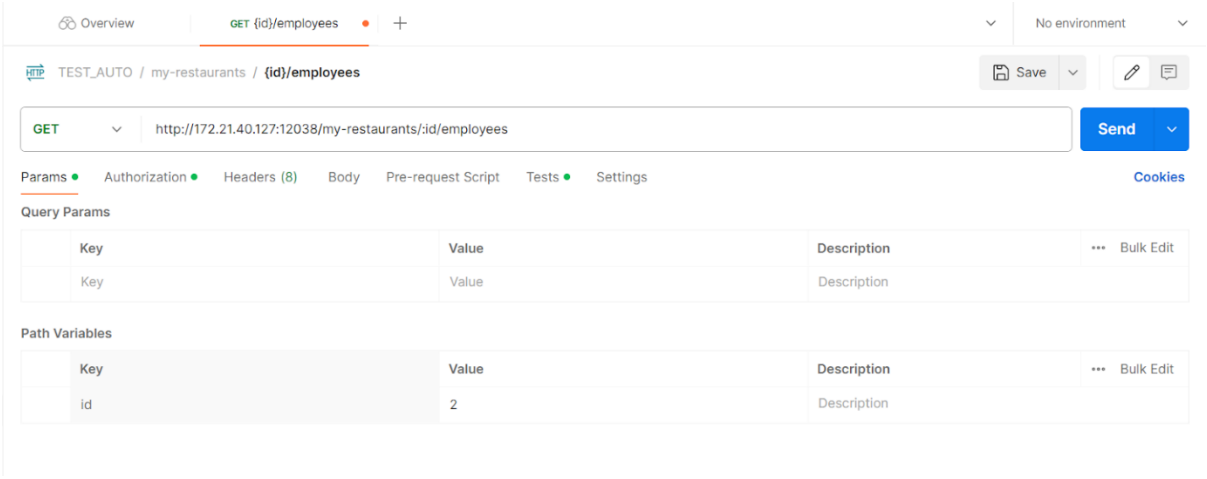
```
            universal_wait_for(driver, EC.element_to_be_clickable, By.ID, "firstName")


            # FIRST NAME
            click_text_field(driver, By.ID, "firstName")


            press_tab_key(driver)


            wait_for_element(driver, By.ID, "firstName-helper-text")


            # LAST NAME
            click_text_field(driver, By.ID, "lastName")


            press_tab_key(driver)


            wait_for_element(driver, By.ID, "lastName-helper-text")
            enter_text(driver, By.ID, "birthDate", RandomData.generate_birth_date())


          # Pozostałe pola


            # Czekamy na pojawienie się przycisku
            # TODO id
            universal_wait_for(driver, EC.element_to_be_clickable, By.CSS_SELECTOR,
"button.pointer:nth-child(2)")


            click_button(driver, By.CSS_SELECTOR, "button.pointer:nth-child(2)")


            # Czekamy na zmianę strony
            universal_wait_for(driver, EC.url_changes, different_value=register_url)


            # Sprawdzenie czy strona zmieniła się na taką jaką chcemy w naszym przypadku
login
            wait_for_url_to_be(driver, login_url)


            return True  # Test przeszedł


        except Exception as e:
            result(str(e), False)
            info("User register test failed")
            info(f"Current URL: {driver.current_url}")
            info(f"Page title: {driver.title}")
            return False  # Test nieudany




        finally:
            wait_for(delay)
```

7.2 Testy końcówek serwera

Testy API zostały przeprowadzone za pomocą narzędzia Swagger, w którym zostały udokumentowane końcówki oraz narzędzi Postman i Bruno. Skorzystano z czarnoskrzynkowej techniki testowania, co oznacza, że testerzy nie byli zaznajomieni z implementacją kodu testowanej funkcjonalności.

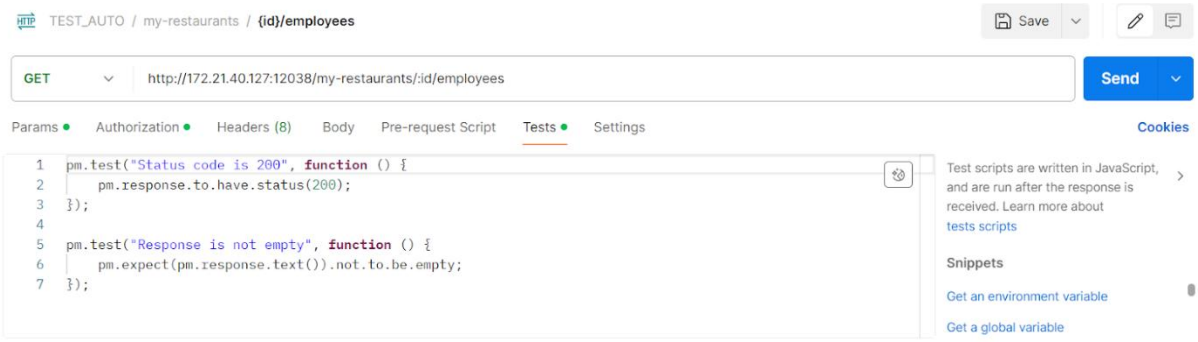
Testy końcówek w aplikacji Postman, były wykonywane poprzez napisanie a następnie wywoływanie funkcji napisanych dla konkretnych końcówek w języku JavaScript. Do testowania były wygenerowane losowe dane zarówno z wykorzystaniem wbudowanych zmiennych "Random" dostępnych w Postmanie jak i z zastosowaniem własnych funkcji generujących losowe dane.

Jako pierwszy, został zaprezentowany test końcówki "*GET/ my-restaurants/:id/employees*", który dla właściciela restauracji wyświetli dane jego pracownika o wybranym id (id jest parametrem, który ustalamy w zakładce "*Params*").



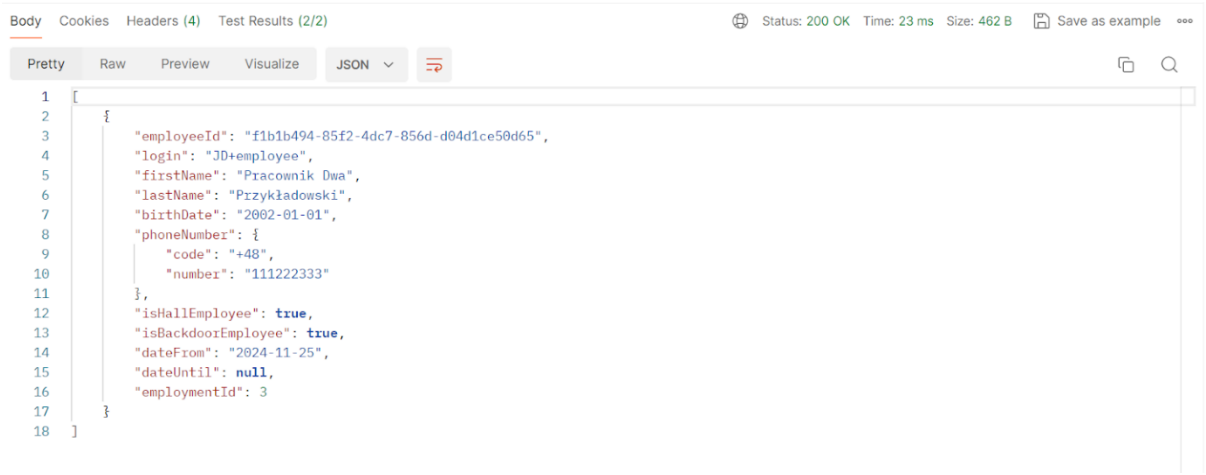
Rysunek 131 Przykład ustawionej końcówki w programie Postman. Źródło: Opracowanie własne

W celu przetestowania końcówki zostały zastosowane zostały wbudowane funkcje programu postman. Wykonanie testu tej końcówki za pomocą narzędzia Postman przedstawia Rysunek 131, Rysunek 132, Rysunek 133 oraz Rysunek 134.



Rysunek 132 Przykład testu końcówki w programie Postman. Źródło: Opracowanie własne

W odpowiedzi otrzymujemy dane pracownika o wybranym ID, oraz informację, że test przeszedł pomyślnie (*status kod - 200 OK*).

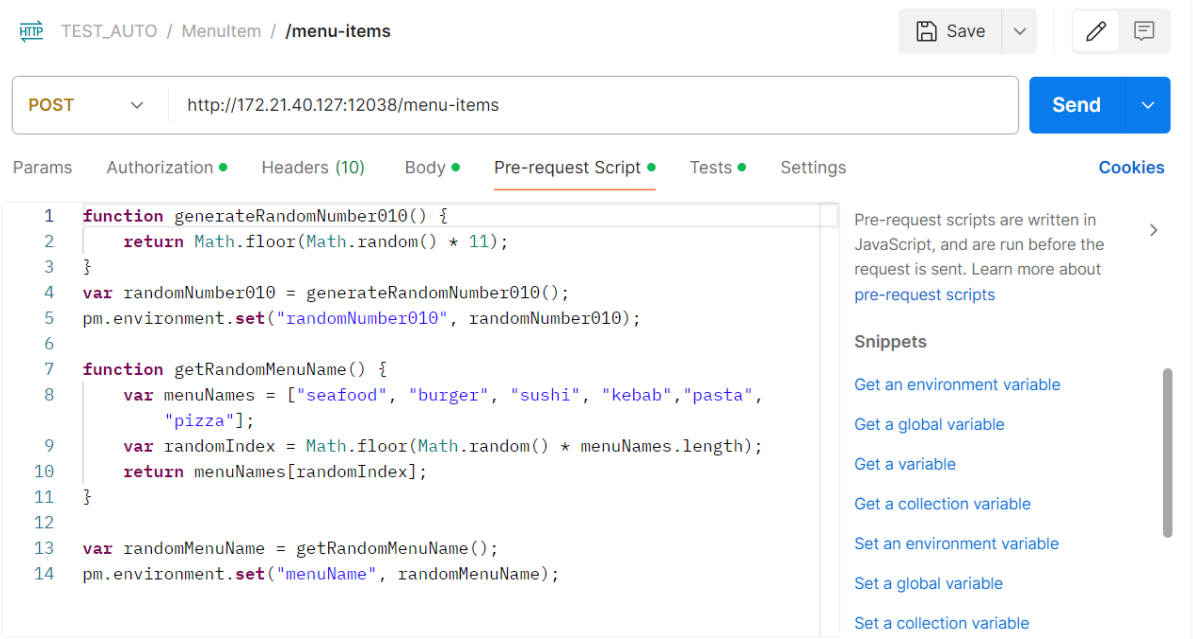


Rysunek 133 Przykład odpowiedzi na zapytanie w programie Postman. Źródło: Opracowanie własne

Rysunek 134 Przykład wyniku testu w programie Postman. Źródło: Opracowanie własne

Kolejną zaprezentowaną końcówką jest "*POST/ menu-items*", która służy do utworzenia nowej pozycji w menu dla wybranej restauracji. Pozycja w menu, utworzona za pomocą tej końcówki będzie mogła następnie zostać dodana do wybranego menu, za pomocą innej końcówki.

W zakładce "*Pre-request Script*" zostały zdefiniowane funkcje służące do wygenerowania losowych wartości z odpowiedniego zakresu. Wygląd tych funkcji przedstawia Rysunek 135.

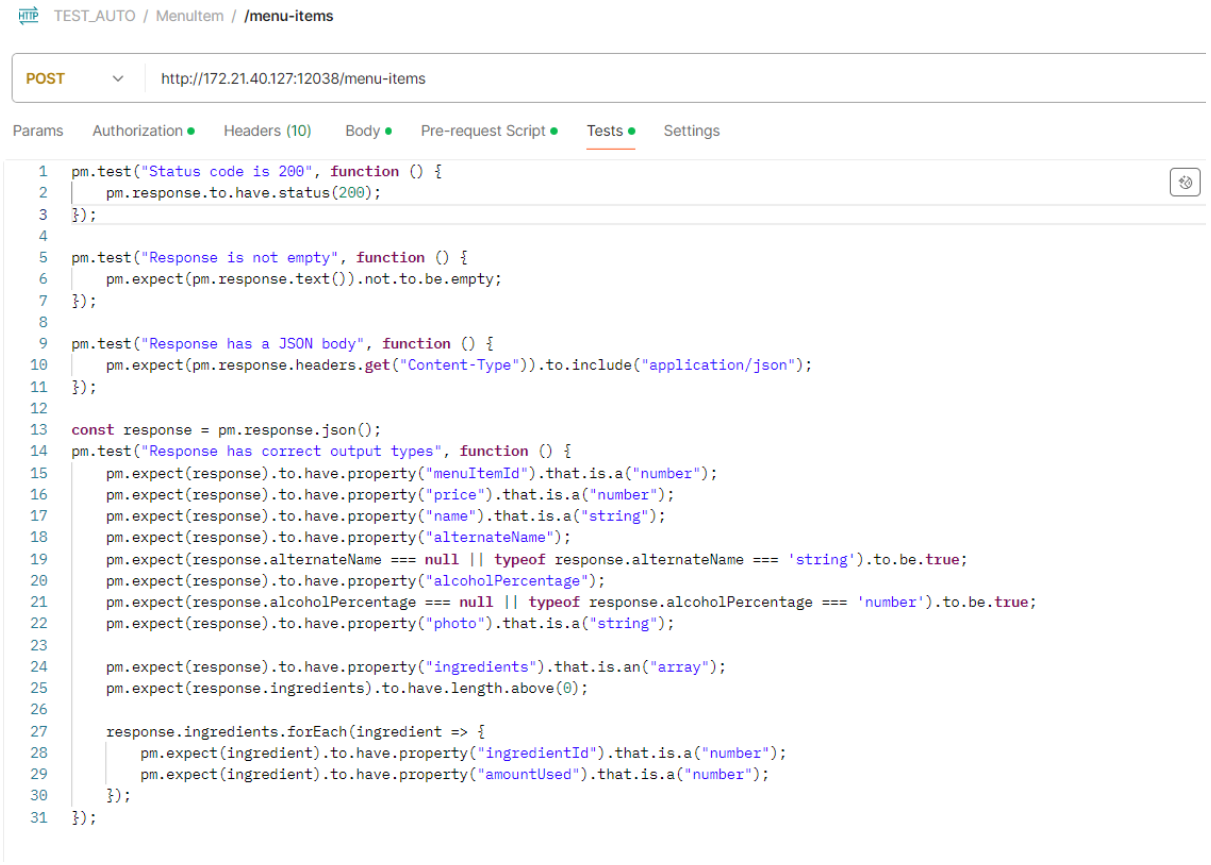


Rysunek 135 Postman końcówka POST /menu-items. Źródło: Opracowanie własne

W zakładce “*Tests*” wykonujemy następujące sprawdzenia:

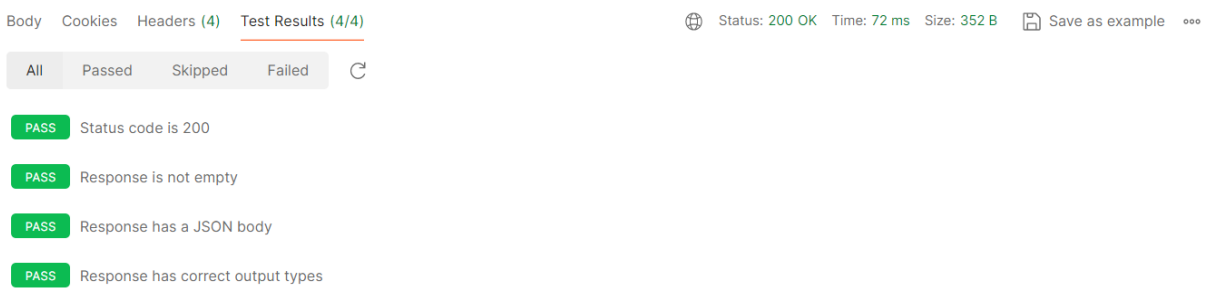
- Sprawdzamy, czy zwracany kod jest kodem 200 OK,
- sprawdzamy, czy otrzymana odpowiedź nie będzie pusta,
- sprawdzamy, czy odpowiedź jest w formacie “JSON”,
- sprawdzamy, czy zwracane wartości są określonych typów (typ numeryczny, tekstowy itd.).

Zawartość zakładki “Test” dla końcówki *POST /menu-items* przedstawia Rysunek 136.



Rysunek 136 Zakładka „Tests" końcówki POST /menu-items. Źródło: Opracowanie własne

W zakładce “*Test Results*” (Rysunek 137) wyświetlana jest informacja, że końcówka, przeszła pomyślnie wszystkie powyższe testy.

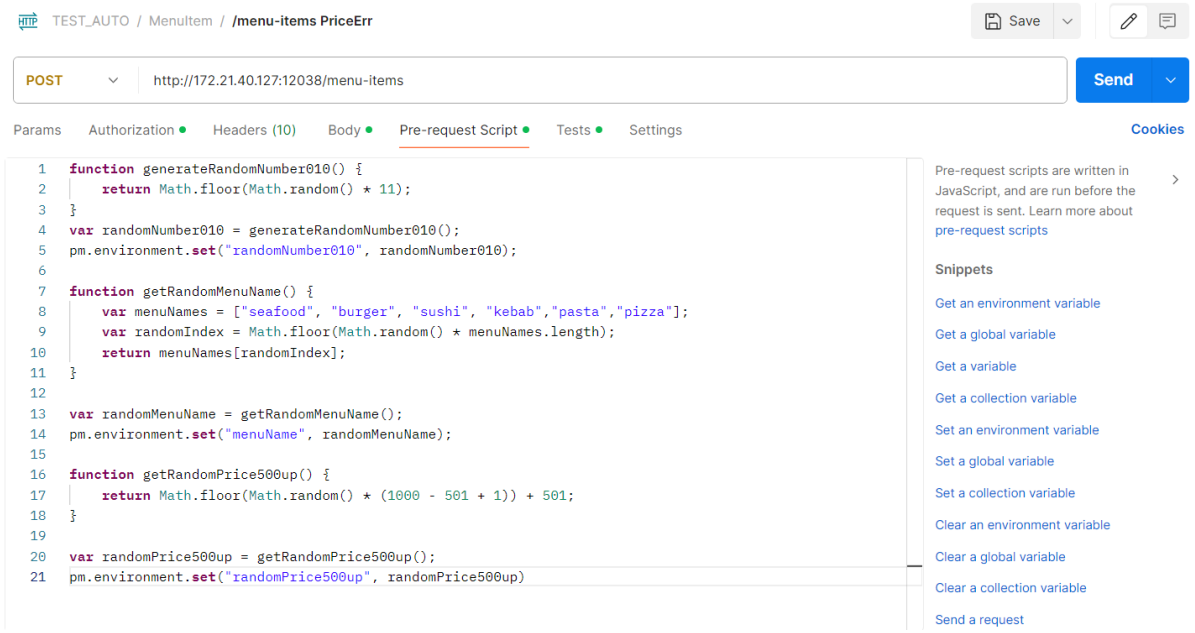


Rysunek 137 Wyniki testu dla końcówki POST /menu-items. Źródło: Opracowanie własne

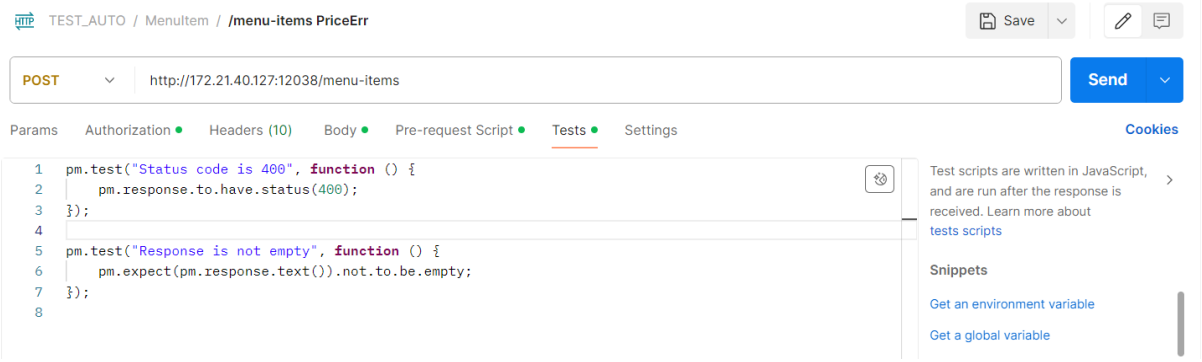
Dla końcówek były wykonywane, także testy negatywne, polegające na celowym wstawieniu złych danych dla końcówki i oczekiwaniu, że błąd ten zostanie wychwycony przez serwer.

Test negatywny "*POST/ menu-items PriceErr*" (Rysunek 139) zostały wykonany dla poprzednio zaprezentowanej końcówki *POST/ menu-items*. Celem tego testu będzie uzyskanie informacji o nieprawidłowo wprowadzonej cenie pozycji z menu. Cena powinna być mniejsza od pięciuset złotych.

Na potrzeby testu, zakładce "*Pre-request Script*" (Rysunek 138) została zdefiniowana funkcja generująca losową liczbę od 500 do 1000.

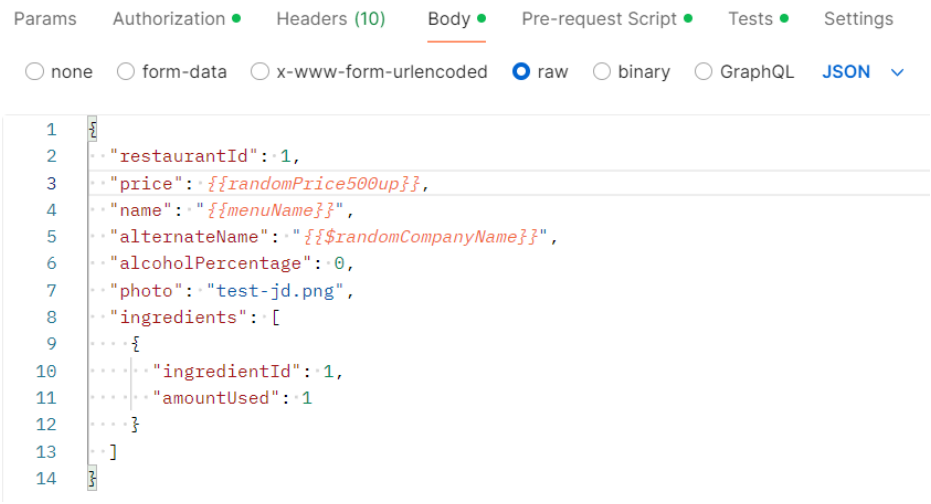


Rysunek 138 Przykład testu PriceErr. Źródło: Opracowanie własne



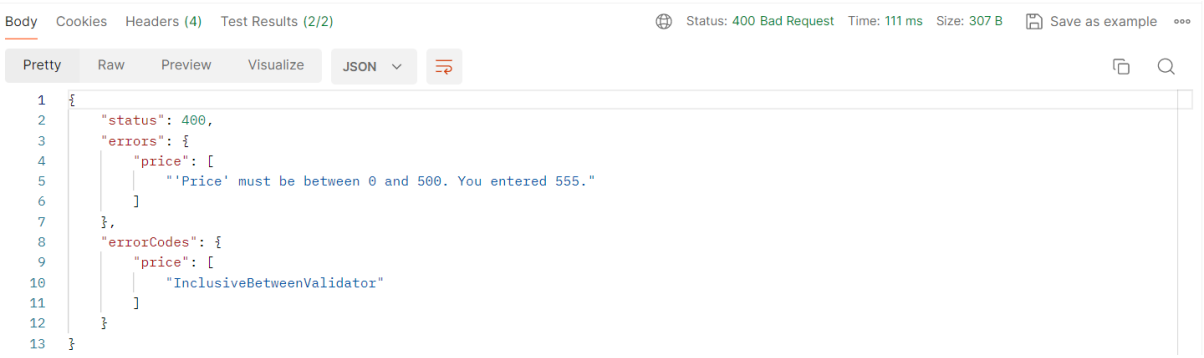
Rysunek 139 Zakładka „Tests" testu PriceErr. Źródło: Opracowanie własne

Następnie liczba ta została wstawiona do wysyłanego metodą post "ciała" (Rysunek 140).



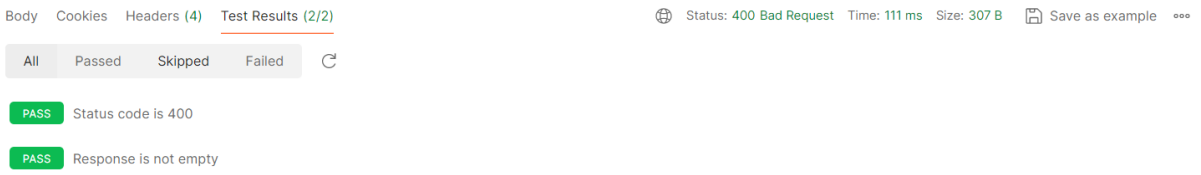
Rysunek 140 Ciało testu PriceErr. Źródło: Opracowanie własne

W odpowiedzi (Rysunek 141) otrzymujemy oczekiwaną informację, o tym, że wartość podana w polu ceny nie jest zgodna z oczekiwaniami.



Rysunek 141 Odpowiedź zapytania dla testu PriceErr. Źródło: Opracowanie własne

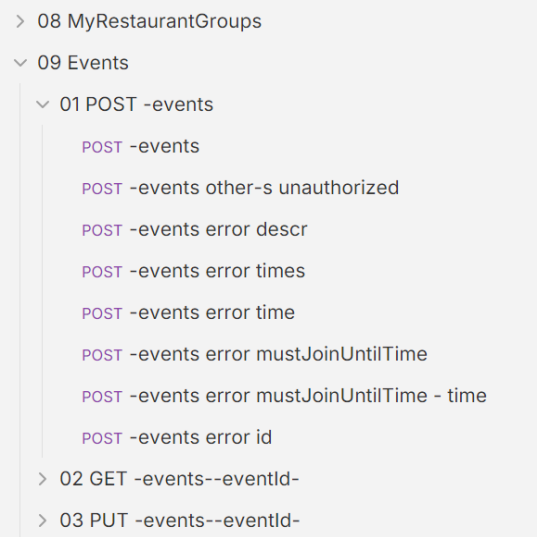
Test przeszedł pomyślnie ze względu na to, że oczekiwaliśmy, że błąd zostanie wychwycony i zwrócony a kod odpowiedzi będzie "*400 niepoprawne zapytanie*", co ukazuje Rysunek 142.



Rysunek 142 Wyniki testu PriceErr. Źródło: Opracowanie własne

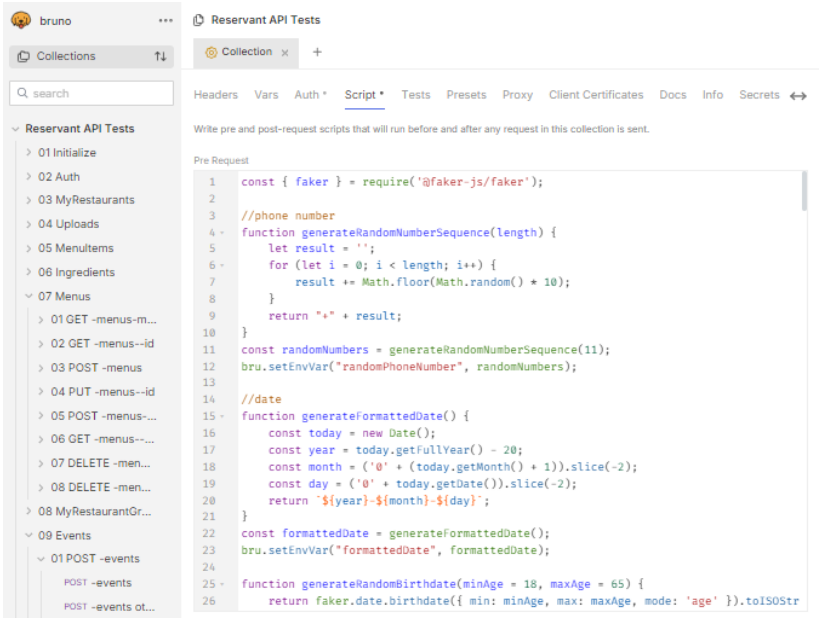
W końcowej fazie projektu testy API zostały wyeksportowane z aplikacji Postman i zaimportowane do aplikacji Bruno API. Bruno w odróżnieniu od Postmana jest open source, co oznacza, że nad testami może współpracować więcej osób oraz umożliwia zarządzanie wersjami przez Github. Z tego względu Bruno został wybrany jako lepsze rozwiązanie dla realizowanego projektu. Oprócz wspomnianych wyżej różnic aplikacje Bruno oraz Postman są bardzo podobne w działaniu i obsłudze.

Testy końcówek zostały umieszczone w konkretnych folderach, odpowiednio dla danej grupy końcówek (np. *Events*) oraz konkretnej końcówki (*GET /events* - pobierającej wszystkie wydarzenia). Przykładową strukturę plików w programie Bruno przedstawia Rysunek 143.



Rysunek 143 Przykładowa struktura plików w programie Bruno. Źródło: Opracowanie własne

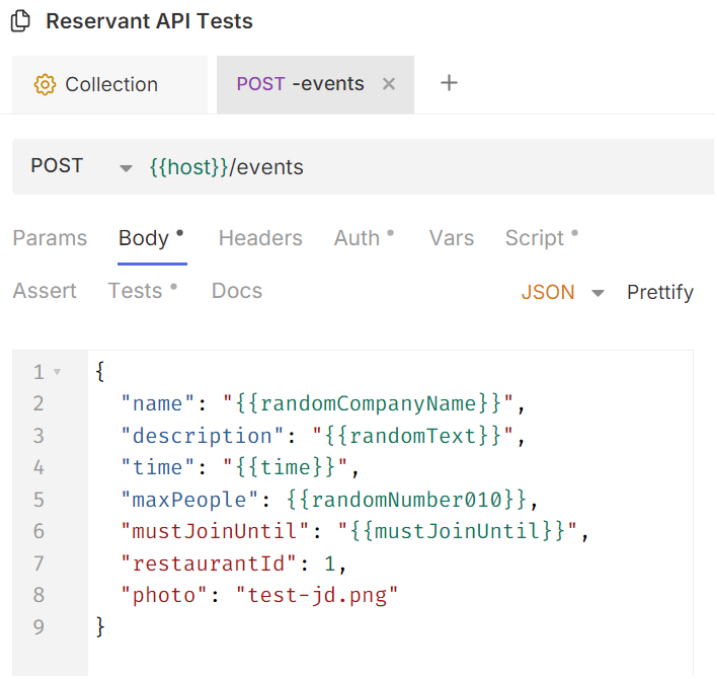
Dla wszystkich końcówek, zarówno jak dla konkretnych grup końcówek i testów konkretnej końcówki, w zakładce "*Script*", zostały zdefiniowane funkcje odpowiadające za generowanie losowych danych potrzebnych do testowania. Na poziomie całego projektu wygenerowane wartości są widoczne z poziomu każdej końcówki, funkcje zdefiniowane na poziomie danej grupy końcówek odpowiednio są widoczne tylko końcówek znajdujących się w danej grupie. Funkcje generujące losowe dane na poziomie konkretnej końcówki, nie mogły być używane dla żadnej innej końcówki, poza tą, dla której zostały zdefiniowane. Wykorzystanie funkcji generującej losowe dane przedstawia Rysunek 144.



Rysunek 144 Przykładowy skrypt w programie Bruno. Źródło: Opracowanie własne

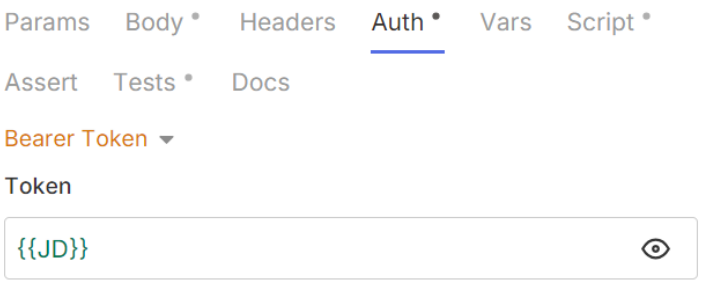
Podobnie jak miało to miejsce w Postmanie, dla końcówek zostały przeprowadzone testy sprawdzające poprawność zwracanych danych.

Rysunek 145 ukazuje przykład testu końcówki „*POST/ events*”. W zakładce “*Body*” generowane są losowe dane, które będą wysyłane dla metody POST.



Rysunek 145 Przykładowe ciało zapytania w programie Bruno. Źródło: Opracowanie własne

Zakładka “*Auth*” (Rysunek 146) służy do uwierzytelnienia użytkownika, który wysyła w danym momencie zapytanie do systemu.



Rysunek 146 Przykład zakładki Auth w programie Bruno. Źródło: Opracowanie własne

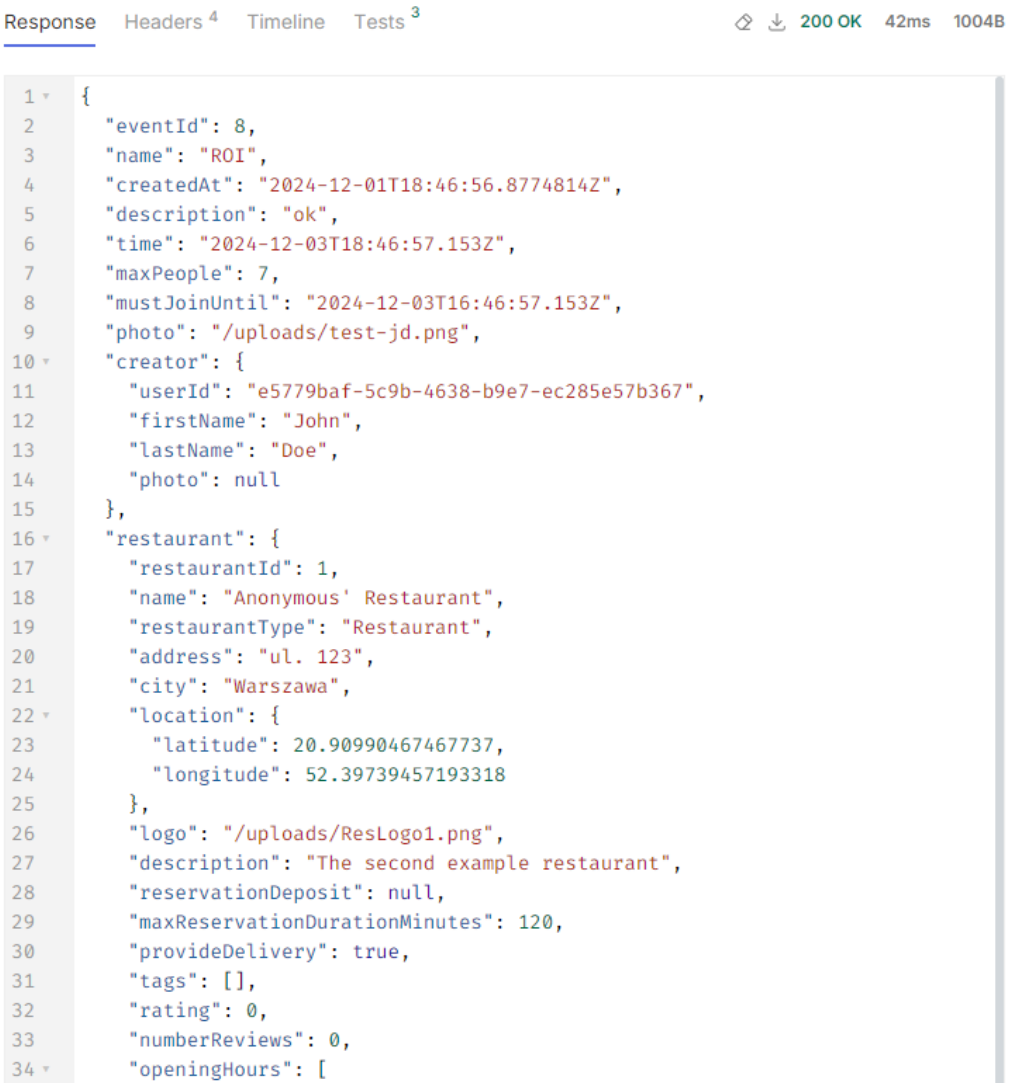
W zakładce “*Tests*” (Rysunek 147) została zdefiniowana treść testu, który przeprowadzany jest na danej końcówce

Params Body * Headers Auth * Vars Script * Assert Tests * Docs

```
test("Status code is 200", function () {
    expect(res.getStatus()).to.equal(200);
});

var jsonData = JSON.parse(JSON.stringify(res.getBody()));

test("Response matches request data", function () {
    var jsonData = res.getBody()
    var sentData = req.getBody()

    expect(jsonData.description).to.eql(sentData.description);
    expect(jsonData.time).to.eql(sentData.time);
    expect(jsonData.mustJoinUntil).to.eql(sentData.mustJoinUntil);
    expect(jsonData.restaurant.restaurantId).to.eql(sentData.restaurantId)
;
});


test("Response has correct data types and required fields", function () {
    var jsonData = res.getBody();

    expect(jsonData).to.have.property("eventId").that.is.a("number");
    expect(jsonData).to.have.property("createdAt").that.is.a("string");
    expect(jsonData).to.have.property("description").that.is.a("string");
    expect(jsonData).to.have.property("time").that.is.a("string");
    expect(jsonData).to.have.property("mustJoinUntil").that.is.a("string")
;
    expect(jsonData).to.have.property("photo").that.is.a("string");
    expect(jsonData).to.have.property("creator").that.is.an("object");
    expect(jsonData).to.have.property("restaurant").that.is.an("object");
    expect(jsonData.restaurant).to.have.property("restaurantId").that.is.a
("number");
    expect(jsonData).to.have.property("participants").that.is.an("array");
});

```

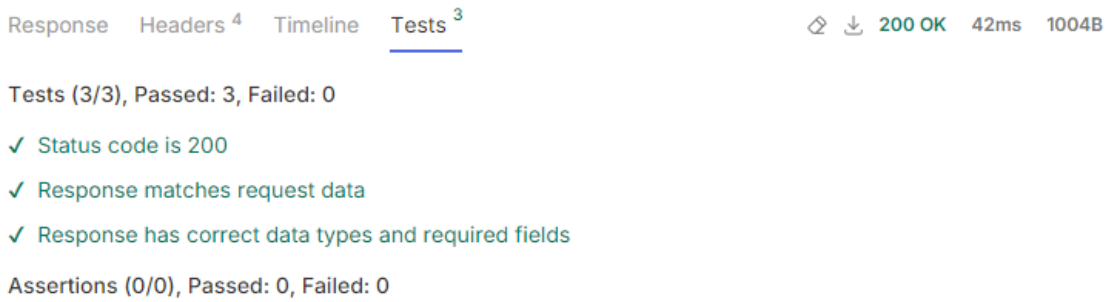
Rysunek 147 Treść przykładowego testu w programie Bruno. Źródło: Opracowanie własne

W zakładce “*Response*” (Rysunek 148) dla końcówki POST, znajdują się losowo wygenerowane dane, które zostały wysłane jako zapytanie.



Rysunek 148 Przykładowa odpowiedź na zapytanie w programie Bruno. Źródło: Opracowanie własne

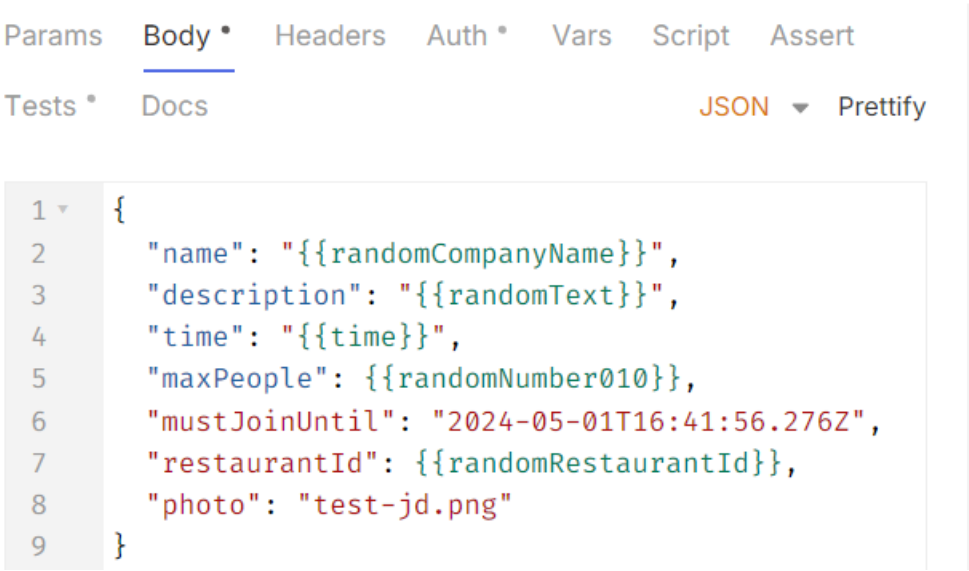
W zakładce "*Tests*" (Rysunek 149) po prawej stronie, widoczna jest informacja, które testy zostały zakończone sukcesem, a które generują błąd podczas wykonywania.



Rysunek 149 Wynik przeprowadzonego testu w programie Bruno

Dodatkowo zostały napisane testy negatywne, w których celowo podaje się złe dane w celu weryfikacji, czy będzie to skutkować błędną odpowiedzią ze strony serwera.

Przykład testu negatywnego zaprezentowany jest dla tej samej końcówki *POST/ events*. Rysunek 150 pokazuje test, który polega na celowym wysłaniu "daty dołączenia do wydarzenia", z przeszłości.

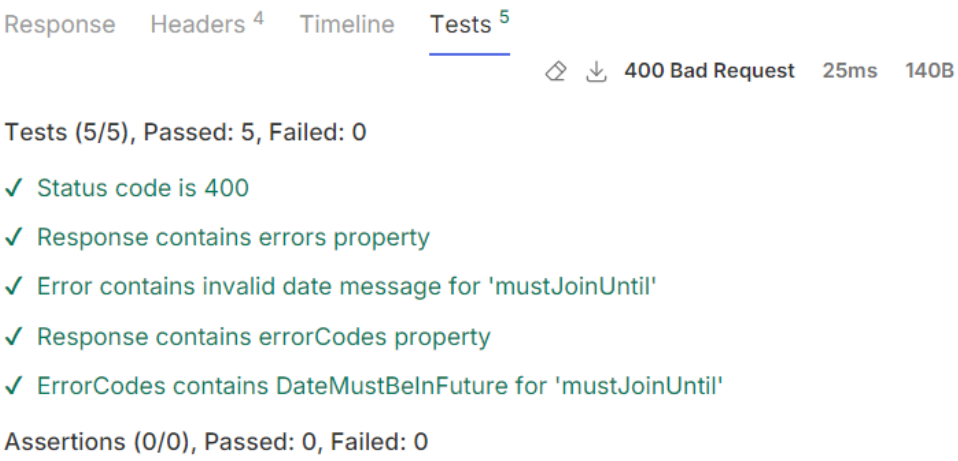


Rysunek 150 Przykład testu mustJoinUntil error. Źródło: Opracowanie własne

Data dołączenia do wydarzenia powinna być datą w teraźniejszości bądź przyszłości i być wcześniejsza niż data wydarzenia. W odpowiedzi otrzymujemy adekwatną informację (Rysunek 151).

Rysunek 151 Przykład odpowiedzi na test mustJoinUntil. Źródło: Opracowanie własne

Wszystkie testy z zakładki test zostały przeprowadzone pomyślnie (Rysunek 152), zgodnie z założeniem otrzymujemy błąd, zapytanie POST nie wysyła błędnych danych.



Rysunek 152 Przykład wyniku testu mustJoinUntil. Źródło: Opracowanie własne

7.3 Testy aplikacji mobilnej

Aplikacja mobilna jest testowana za pomocą testów jednostkowych. Głównym celem testów jest sprawdzenie czy obsługiwane są w poprawny sposób wszystkie końcówki jakie używa aplikacja w komunikacji z serwerem. W tym celu wykorzystano możliwości wykonywania testów bezpośrednio w android studio. Program posiada wbudowany moduł odpowiedzialny za wykonywanie testów połączonych z kontekstem emulatora. Jest to kluczowe w przypadku omawianych testów ze względu na specyfikę ich działania oraz konieczność połączenia z siecią.

Schemat wykonywania testów wielokrotnie się powtarza co może doprowadzić do niepotrzebnej redundancji kodu. W celu uniknięcia takiego stanu rzeczy utworzono tutaj specjalną klasą abstrakcyjną "`ServiceTest`" (Listing 73), która implementuje metody wykonywujące się przed oraz po każdym teście.

Listing 73 Implementacja klasy ServiceTest. Źródło: Opracowanie własne

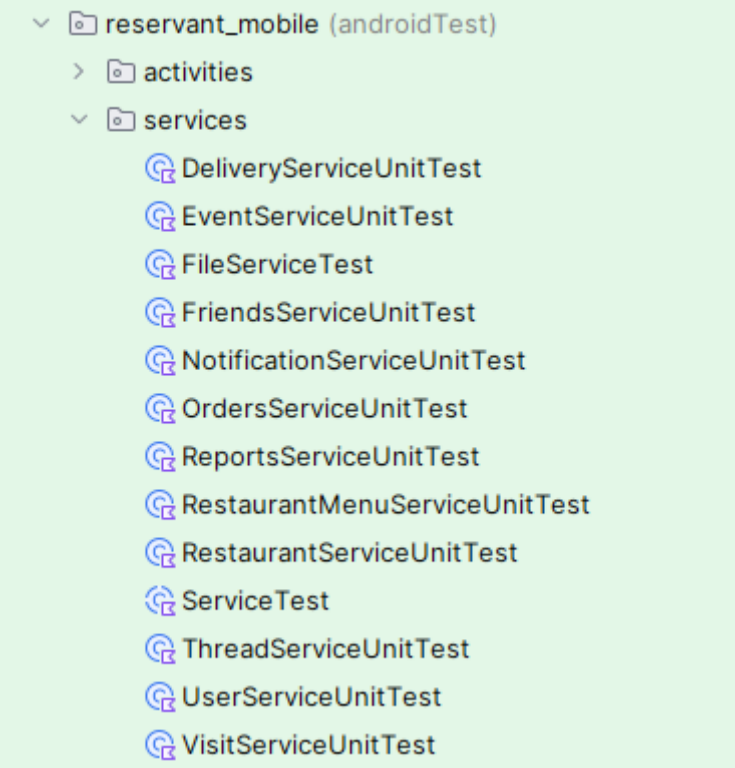
```
abstract class ServiceTest {
    private lateinit var existingToken: String
    protected val localBearer = LocalDataService()
    protected val userService: IUserService = UserService()
    protected var loginUser: LoginCredentialsDTO = LoginCredentialsDTO(
                                                    login = "JD",
                                                    password = "Pa${"$"}${"$"}w0rd",
                                                    rememberMe = false
                                                    )
    @Before
    fun getPreTestData() = runBlocking {
        existingToken = localBearer.getData(PrefsKeys.BEARER_TOKEN)
    }


    @After
    fun cleanPostTestData() = runBlocking{
        userService.logoutUser()
        if(existingToken.isNotEmpty())
            localBearer.saveData(PrefsKeys.BEARER_TOKEN,existingToken)
    }


    suspend fun loginUser(user: LoginCredentialsDTO = loginUser){
        userService.loginUser(user)
    }
}
```

Testy zostały podzielone pod względem tematyki danego serwisu. Dlatego przykładowo `ThreadsService` będzie posiadał swój odpowiednik w postaci `ThreadServiceUnitTest` (Listing 74) itd. Większość metod danego serwisu zostaje pojedynczo sprawdzana.

Android studio samodzielnie wykrywa utworzone testy w przypadku, gdy spełnione są następujące warunki:

- Moduł klasy z testami umieszczony jest w folderze "*androidTest*",
- Każda z metod testujących posiada adnotację "`@Test`".

Gdy warunki te zostaną spełnione, testy mogą być wykonywane pojedynczo lub automatycznie wszystkie na raz. Sposób organizacji klas testujących serwisy przedstawia Rysunek 153.



Rysunek 153 Implementacja klas testujących serwisy. Źródło: Opracowanie własne

Dzięki temu, że każda z klas dziedziczy po klasie abstrakcyjnej `ServiceTest` wszystkie testy są odpowiednio skonfigurowane oraz zostawiają po sobie porządek w postaci wyczyszczenia danych logowania. Dzieje się to za sprawą metod, które posiadają odpowiednie adnotacje. W tym przypadku "`@Before`" oznacza, że dana metoda zostanie wykonana przed każdym testem. Adekwatnie "`@After`" będzie wykonane, gdy test się zakończy.

Omawiane testy nie skupiają się na tym jakie dane zostanę przysłane przez serwer. Ich celem jest sprawdzenie poprawności działania serializatora oraz konfiguracji danej końcówki. Warto tutaj również zwrócić uwagę na fakt, że testy sprawdzają również poprawność obsługi paginacji.

Listing 74 Implementacji klasy testującej "ThreadServiceUnitTest". Źródło: Opracowanie własne

```
class ThreadServiceUnitTest:ServiceTest() {
    private val ser: IThreadsService = ThreadsService()
    private val jdId = "e5779baf-5c9b-4638-b9e7-ec285e57b367"
    private val customerId = "e08ff043-f8d2-45d2-b89c-aec4eb6a1f29"




    @Before
    fun setupData() = runBlocking {
        loginUser()
    }


    @Test
    fun create_and_delete_thread()= runTest{
        val res = ser.createThread(
            title = "Test thread",
            participantIds = listOf(jdId, customerId)
        ).value
        assertThat(res).isNotNull()
        assertThat(ser.deleteThread(res!!.threadId!!).value).isTrue()
```

```
    }


    @Test
    fun edit_thread_return_not_null()= runTest{
        val thread = ThreadDTO(
            title = "Test Thread Title"
        )
        assertThat(ser.editThread(1, thread).value).isNotNull()
    }


    @Test
    fun get_thread_return_not_null()= runTest{
        assertThat(ser.getThread(1).value).isNotNull()
    }


    @Test
    fun create_and_delete_message()= runTest{
        val res = ser.createMessage(1, "test message").value
        assertThat(res).isNotNull()
        assertThat(ser.deleteThread(res!!.messageId!!).value).isNotNull()
    }


    @Test
    fun get_messages_return_pagination()= runTest{
        val items = ser.getMessages(1).value
        val itemsSnapshot = items?.asSnapshot {
            scrollTo(index = 10)
        }
        assertThat(itemsSnapshot).isNotEmpty()
    }


    @Test
    fun edit_message_return_not_null()= runTest{
        assertThat(ser.editMessage(1, "message").value).isNotNull()
    }


    @Test
    fun mark_message_as_read_return_not_null()= runTest{
        assertThat(ser.markMessageAsRead(1).value).isNotNull()
    }
}
```

7.4 Wykonywanie testów

Wszystkie opisane w tym rozdziale testy są wykonywane na różne sposoby. Większość uruchamiana jest w sposób zautomatyzowany. Niektóre z nich wymagały jednak manualnego podejścia, które zapewniłoby merytoryczną ocenę danego elementu przez osobę testującą.

7.4.1 Testy zautomatyzowane

W rozdziale 5.2 omówiono automatyczną kompilację kodu podczas jego publikacji. Ustawiono tam również instrukcje odpowiadające za zautomatyzowane wykonywanie testów. Dobrym przykładem jest chociażby sposób w jaki zaimplementowano testy serwera backendowego. W głównym katalogu repozytorium stworzono tam plik "*Jenkinsfile.tests*" oraz katalog "test" zawierający testy wszystkich końcówek wraz z całą konfiguracja i plikiem "*Dockerfile.tests*". Listing 75 prezentuje plik odpowiedzialny za budowanie obrazu dockera, który tworzy wyodrębnione środowisko przeprowadzające testy w programie bruno.

Listing 75 Przykład pliku Dockerfile.tests. Źródło: Opracowanie własne

```
FROM node:16

WORKDIR /app

COPY ./test .
ENV CI=true
RUN npm install -g @usebruno/cli@1.35
RUN npm install

ENV POST_MSG_HEAD=""
ENV DISCORD_WEBHOOK_URL=""

RUN chmod +x ./run_tests.sh
CMD ["/bin/bash", "-c", "./run_tests.sh ${DISCORD_WEBHOOK_URL} ${POST_MSG_HEAD}"]
```

W ten sposób otrzymano gotowe środowisko, które pozwoli wykonać testy na dowolny urządzeniu, na którym zainstalowany jest docker. Plik, który prezentuje Listing 76 jest odpowiedzialny za wykonywanie odpowiednich instrukcji przez system Jenkins. Przygotowano tam potok, który wykonywany jest w dwóch przypadkach:

5. Okresowo, testy wykonywane są trzy razy w tygodniu co stanowi sprawdzanie stabilności systemu podczas wprowadzania każdorazowo losowych danych,
6. Podczas publikacji kodu serwera backendowego, co pozwala testować system pod kątem regresyjnym.

Listing 76 Przykład pliku Jenkinsfile.tests. Źródło: Opracowanie własne

```
pipeline {
    agent any

    options {
        disableConcurrentBuilds()
    }

    parameters {
        string(name: "label_string", defaultValue: "(PERIODIC)", trim: true,
description: "Sample string parameter")
    }
```

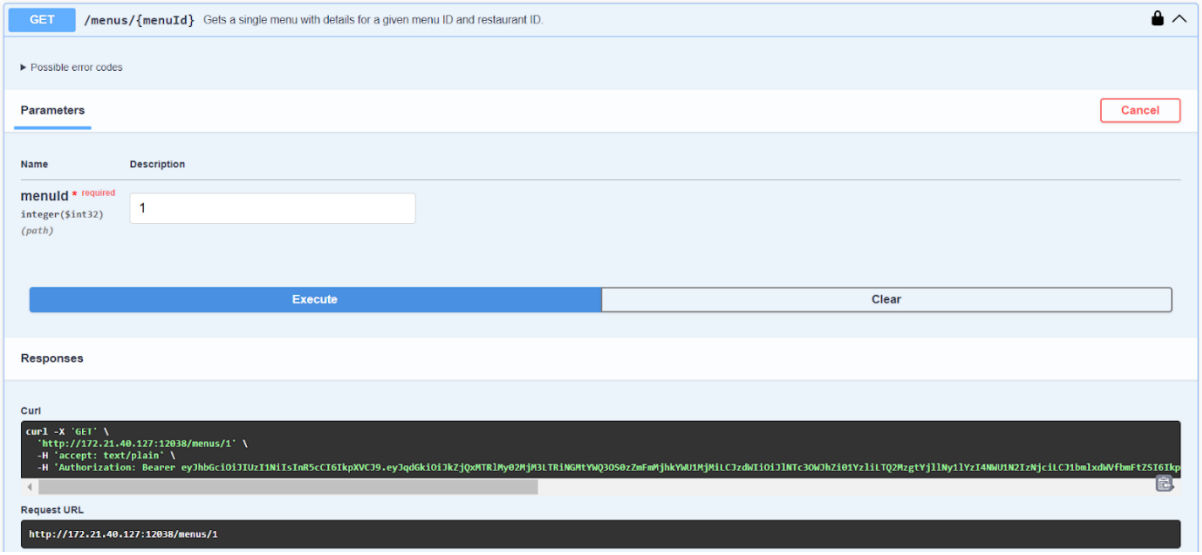
```
        environment {
            DISCORD_WEBHOOK_URL = credentials('jenkins-back-tests-discord-webhook')
            INFO_LABEL = "${params.label_string}"
        }

        stages {
            stage('Build test image') {
                steps {
                    sh "ls -al"
                    sh "docker build -t reservant-backend-tests -f test/Dockerfile.tests ."
                }
            }
            stage('Run tests') {
                steps {
                    sh "docker run -e DISCORD_WEBHOOK_URL=$DISCORD_WEBHOOK_URL -e
POST_MSG_HEAD='$INFO_LABEL' reservant-backend-tests"
                }
            }
        }
    }
```

Warto tutaj również wspomnieć, że testy aplikacji webowej są wykonywane w bliźniaczo podobny sposób z tą różnicą, iż zamiast tworzyć folder w tym samym repozytorium, stworzono oddzielne, kompletnie nowe repozytorium.

7.4.2 Testy Manualne

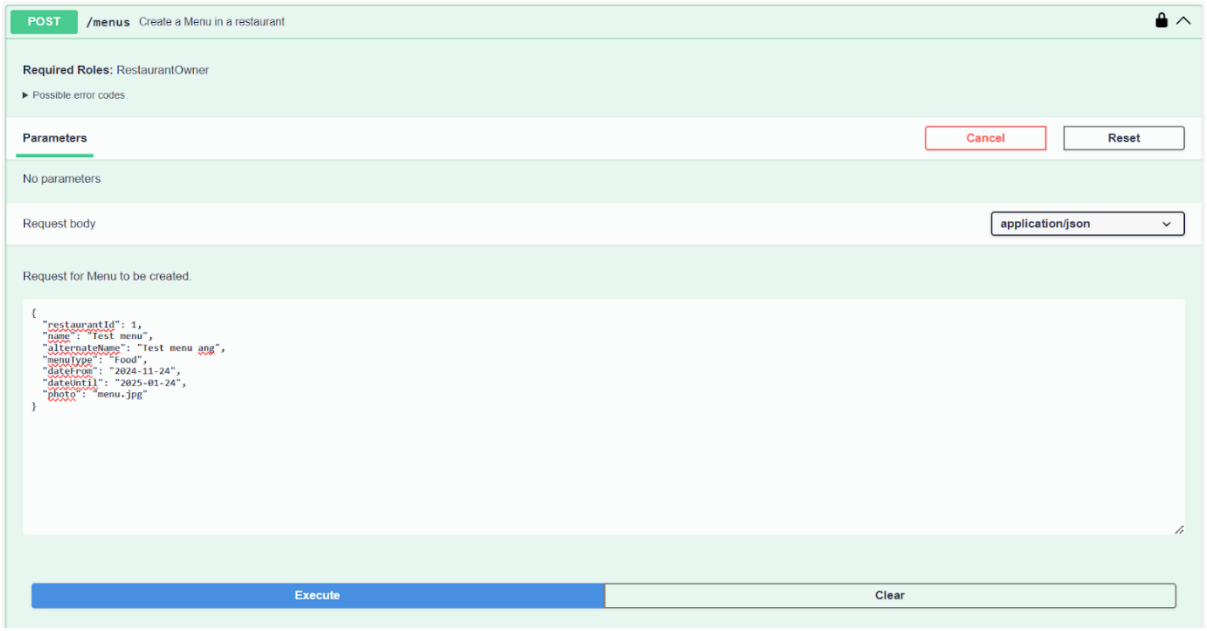
Niektóre elementy systemu należało jednak sprawdzać własnoręcznie. Uzgodniono, że poszczególne aspekty systemu warto dodatkowo testować manualnie w celu oceny pod kątem wygody użytkowania i ogólnego doświadczenia z aplikacją.

Narzędzie Swagger pozwoliło w łatwy sposób udokumentować oraz jednocześnie ręcznie testować końcówki jakie zostały udostępnione przez serwer. Graficzny interfejs użytkownika umożliwia wykonanie danej końcówki wraz z obsługą wymaganych parametrów. Rysunek 154 oraz Rysunek 156 prezentują wywołanie przykładowych końcówek GET oraz POST. Rysunek 155 przedstawia odpowiedź serwera na wysłane żądanie.



Rysunek 154 Wywołanie końcówki GET, Swagger. Źródło: Opracowanie własne

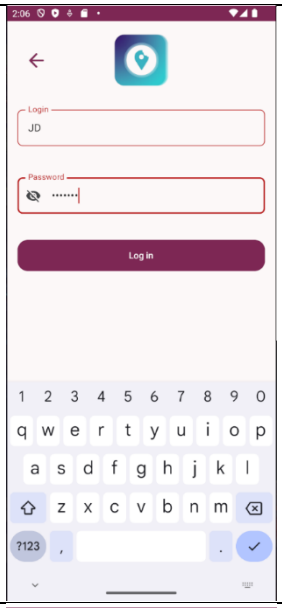
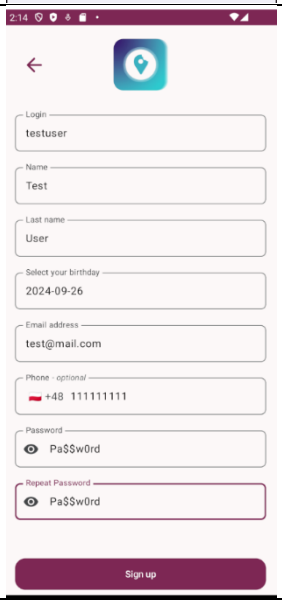
Rysunek 155 Odpowiedź końcówki GET, Swagger. Źródło: Opracowanie własne

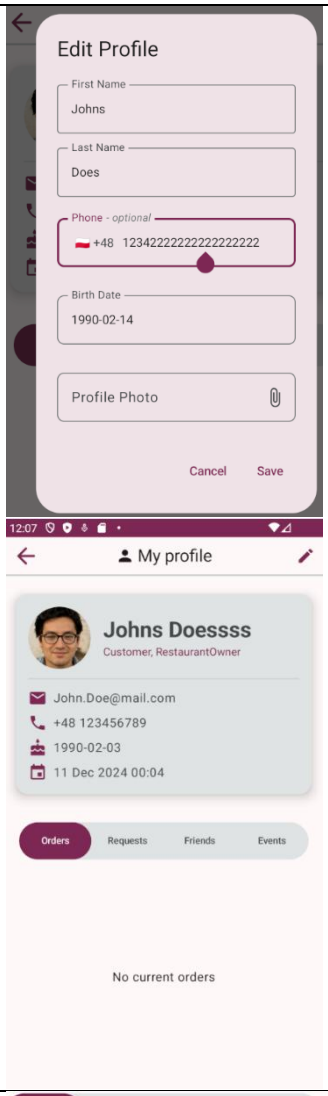
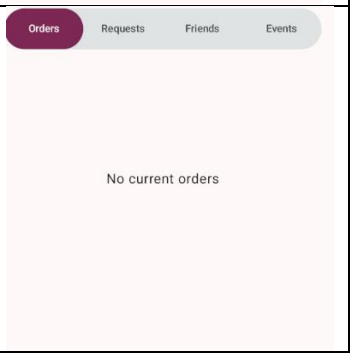


Rysunek 156 Wywołanie końcówki POST, Swagger. Źródło: Opracowanie własne

Ręcznie testowana była również aplikacja mobilna, ponieważ ograniczenia techniczne nie pozwoliły na implementację zautomatyzowanego potoku testującego interfejs użytkownika. W tym celu zespół testów przygotował strukturę raportu, który był uzupełniany regularnie przez osoby testujące. Tabela 16 pokazuje przykład takiego raportu dla kilku przypadków użycia. Takie podejście pozwoliło na ustrukturyzowany podział pracy i jasny wgląd do jej efektów.

Tabela 16 Przykład raportu testu manualnego. Źródło: Opracowanie własne

Testowany ekran	Testowane funkcjonalności	Znalezione nieprawidłowości	Zrzut ekranu
Logowanie	Logowanie błędnymi danymi	-	-
	Logowanie poprawnymi danymi	-Nie pojawiają się ostrzeżenia o np. braku loginu albo hasła - nie pojawia się żadne powiadomienie typu „Nieprawidłowe hasło” czy „Podany adres email nie istnieje” - Przycisk powracania nic nie robi	
Rejestracja użytkownika	Rejestracja użytkownika	-Serwer zwraca błąd o niepoprawnym formacie numeru telefonu, ale na ekranie nie wyświetla się żaden błąd. Po prostu nie działa przycisk ‘Sign up’ - Przycisk powracania nic nie robi	
Restaurant management	Edycja danych restauracji	Potwierdzenie edycji restauracji zamknęło aplikacje	
Chats	Wyświetlanie szczegółów czatu	-	-
	Wyszukiwanie chatu po nazwie	Funkcjonalność wyszukiwania chatu po nazwie jest case sensitive (a raczej nie powinna? Np. wpisując w search „example” nie wyświetla threadów o	

		nazwie „Example Thread”	
Settings > My Profile > Edycja danych	Edycja danych użytkownika	-Wprowadzenie błędnego numeru telefonu i kliknięcie Save nie pokazuje żadnych błędów ostrzeżeń - ‘Save’ w ogóle nic nie robi? Logi backendu pokazują, że request się dobrze wysyła, ale formularz edycji nie znika i dopiero po przejściu na inny ekran i z powrotem na My profile nazwa użytkownika się odświeża (drugi screen)	
Settings > My Profile	Przeglądanie zakładek	Jeżeli user nie ma żadnych requestów, to warto by było wyświetlać tekst typu ‘No requests’ a nie pusty komponent (tak jak w Orders, drugi screen)	

8 Podsumowanie

Prace nad projektem były wymagające, zarówno pod względem czasu, jak i zaangażowania zespołu. Proces tworzenia oprogramowania od podstaw, obejmujący analizę, projektowanie, implementację i testowanie, trwał od kilku miesięcy do trzech semestrów nauki, z uwzględnieniem przerw świątecznych i wakacyjnych. Projekty realizowały grupy złożone z kilku osób o różnym poziomie wiedzy i doświadczenia w zakresie inżynierii oprogramowania oraz kompetencji miękkich.

Podczas realizacji pojawiały się różnorodne trudności, takie jak problemy z komunikacją, nierównomierne rozłożenie obowiązków, brak motywacji niektórych członków zespołów, a także wyzwania organizacyjne związane z różnorodnością umiejętności i doświadczenia. Zespoły mierzyły się także z koniecznością reagowania na zmiany personalne i w strukturach kierowniczych. Dzięki systematycznej analizie problemów i podejmowaniu działań naprawczych udało się jednak zakończyć prace zgodnie z założeniami.

Udział w projektach pozwolił uczestnikom na zdobycie cennego doświadczenia w realizacji kompleksowych przedsięwzięć programistycznych. Wielu z nich nabyło nowe umiejętności, które mogą się przyczynić do rozwoju zawodowego.

8.1 Efekty pracy

Aplikacja Reservant miała na celu stworzenie kompleksowego rozwiązania informatycznego w postaci serwera, aplikacji webowej i mobilnej. Po zapoznaniu się z dostępnymi na rynku aplikacjami oraz wnikliwej analizie obszarów, w których Reservant mógłby wprowadzać innowacje w porównaniu do konkurencji, zespół analityczny rozpoczął prace nad funkcjonalnościami aplikacji.

Głównym założeniem projektu było ułatwienie interakcji między klientami a właścicielami restauracji poprzez wdrożenie odpowiednich funkcjonalności. Reservant pozwala klientom na wygodne przeglądanie oferty lokali gastronomicznych, rezerwowanie stolików oraz składanie zamówień na wybrane dania, co jest możliwe dzięki centralnej roli serwera, obsługującego te funkcje w czasie rzeczywistym.

Klienci mogą również dodawać znajomych, komunikować się z nimi oraz wystawiać opinie restauracjom. Z kolei właściciele restauracji zyskują narzędzie umożliwiające dodawanie swoich lokali do platformy, zarządzanie rezerwacjami i pracownikami, aktualizację menu, monitorowanie statystyk, kontrolowanie stanu magazynu, przeglądanie opinii i zgłoszeń od klientów oraz śledzenie zrealizowanych dostaw. Serwer pełni tutaj rolę centralnego miejsca zarządzania tymi danymi, umożliwiając ich sprawną synchronizację między wszystkimi użytkownikami systemu.

Podczas prac zespołu implementacyjnego, a także po ich zakończeniu, stale przeprowadzano testy zarówno aplikacji webowej, jak i mobilnej. Testy obejmowały również serwer i jego funkcjonalności w celu zapewnienia stabilności systemu. Zespół odpowiedzialny za testy monitorował proces oraz wykonywał szereg działań, które poskutkowały wyeliminowaniem licznych błędów, które pojawiły się podczas intensywnych prac zespołu implementacji.

W trakcie pracy nad projektem zespoły regularnie napotykały nowe problemy, których rozwiązywanie znacznie poszerzyło wiedzę i doświadczenie studentów zaangażowanych w projekt. Jednym z najważniejszych efektów tej pracy jest zdobyte doświadczenie, które zespoły wyniosły ze wspólnej współpracy.

8.2 Napotkane problemy

W czasie prac nad projektem zespół napotkał rozmaite trudności z różnych dziedzin. Niniejszy rozdział został podzielony na podrozdziały, w których opisano trudności występujące w różnych zespołach.

Trudności, które pojawiły się w zespole, są charakterystyczne dla liczniejszych grup projektowych. Do problemów tych należą m.in. komunikacja oraz dotrzymywania ustalonych terminów przez wszystkich członków zespołu. W odpowiedzi na te problemy z pomocą przyszła aplikacja Trello, służąca do zarządzania zadaniami, wyznaczania zadaniom określonych terminów wykonania oraz pozwalająca na przypisanie zadań do konkretnych członków zespołu. W rozwiązaniu wspomnianych trudności, pomocna była również aplikacja Discord, na której można było m.in zaplanować spotkania, dla wybranych zespołów w projekcie.

8.2.1 Analiza i projektowanie

W trakcie realizacji analizy projektu pojawiły się problemy charakterystyczne dla tego etapu pracy, które wpłynęły na organizację i spójność opracowywanych materiałów.

Jednym z głównych wyzwań było realizowanie zależnych od siebie zadań w sposób nieuporządkowany, co skutkowało niespójnościami pomiędzy różnymi diagramami. Przykładowo, prace nad diagramem klas (CLD) oraz diagramem przypadków użycia, (UD) i opisem funkcjonalnym były prowadzone równocześnie przez różne zespoły. W efekcie każdy z tych „sektorów” był stale zmieniany, co utrudniało ich synchronizację.

Kolejną trudnością była organizacja zadań w systemie Trello. Niektórzy członkowie zespołu nie korzystali z narzędzia w ogóle, co wymuszało indywidualne przypominanie o zadaniach, co było sprzeczne z ideą używania tego typu platformy. Inni niepoprawnie korzystali z tablicy – na przykład, zamiast przekazywać zadań do kolumny „*Review*” do sprawdzenia, wrzucali je bezpośrednio do „*Done*”. Powodowało to chaos w organizacji pracy i wymagało dodatkowej weryfikacji zadań.

Dodatkowym problemem było przypisywanie zadań przez samych członków zespołu. Tego rodzaju samodzielność prowadziła do nieprzewidzianych konfliktów i trudności w zarządzaniu zespołem. W niektórych przypadkach członkowie zespołu zgłaszali swoje nieobecności (np. wyjazdy, choroby) zbyt późno, co uniemożliwiało efektywne rozdzielenie ich zadań między pozostałe osoby.

Jednym z głównych wyzwań był również brak wystarczających kompetencji członków zespołu. Przedmiot, który miał dostarczyć odpowiednią wiedzę na temat prawidłowego tworzenia diagramów, był realizowany równolegle z implementacją projektu. W rezultacie zespół musiał stworzyć diagramy bez pełnego zrozumienia zasad ich poprawnej konstrukcji. Doprowadziło to do powstania wielu elementów, których spójność z backendem była niepewna.

Kolejną kwestią było wykonywanie zadań bez pełnego zrozumienia projektu. Niektórzy członkowie zespołu, z braku ogólnej wiedzy, realizowali zadania intuicyjnie, co skutkowało koniecznością ich poprawy. Problem ten wynikał głównie z niewystarczającej organizacji pracy na wstępnym etapie projektu oraz braku odpowiedniej komunikacji, mimo dostępności narzędzi dedykowanych temu celowi.

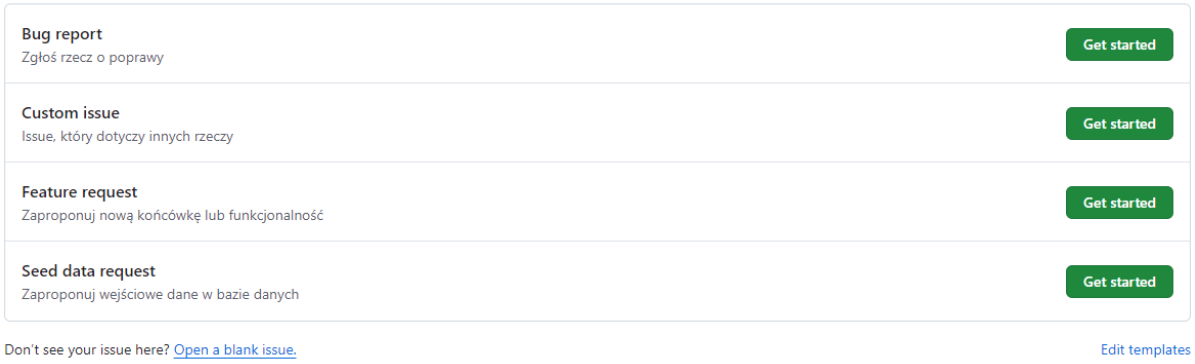
8.2.2 Backend

Jednym z problemów napotkanych podczas implementacji serwera była potrzeba ręcznego przepisywania danych między obiektami modelu (reprezentującymi dane w bazie) a obiektami DTO (ang. *Data Transfer Objects*), które służą do przekazywania danych między warstwami aplikacji. Taka konwersja była wymagana zarówno w kodzie C#, jak i w zapytaniach LINQ (ang. *Language Integrated Query*), które Entity Framework tłumaczy na SQL.

Aby uprościć ten proces i uniknąć redundancji, zastosowano bibliotekę AutoMapper. Pozwala ona zdefiniować reguły mapowania w jednym miejscu, co automatyzuje przekształcanie danych między modelami a DTO. Dzięki temu kod stał się bardziej czytelny, mniej podatny na błędy i łatwiejszy w utrzymaniu.

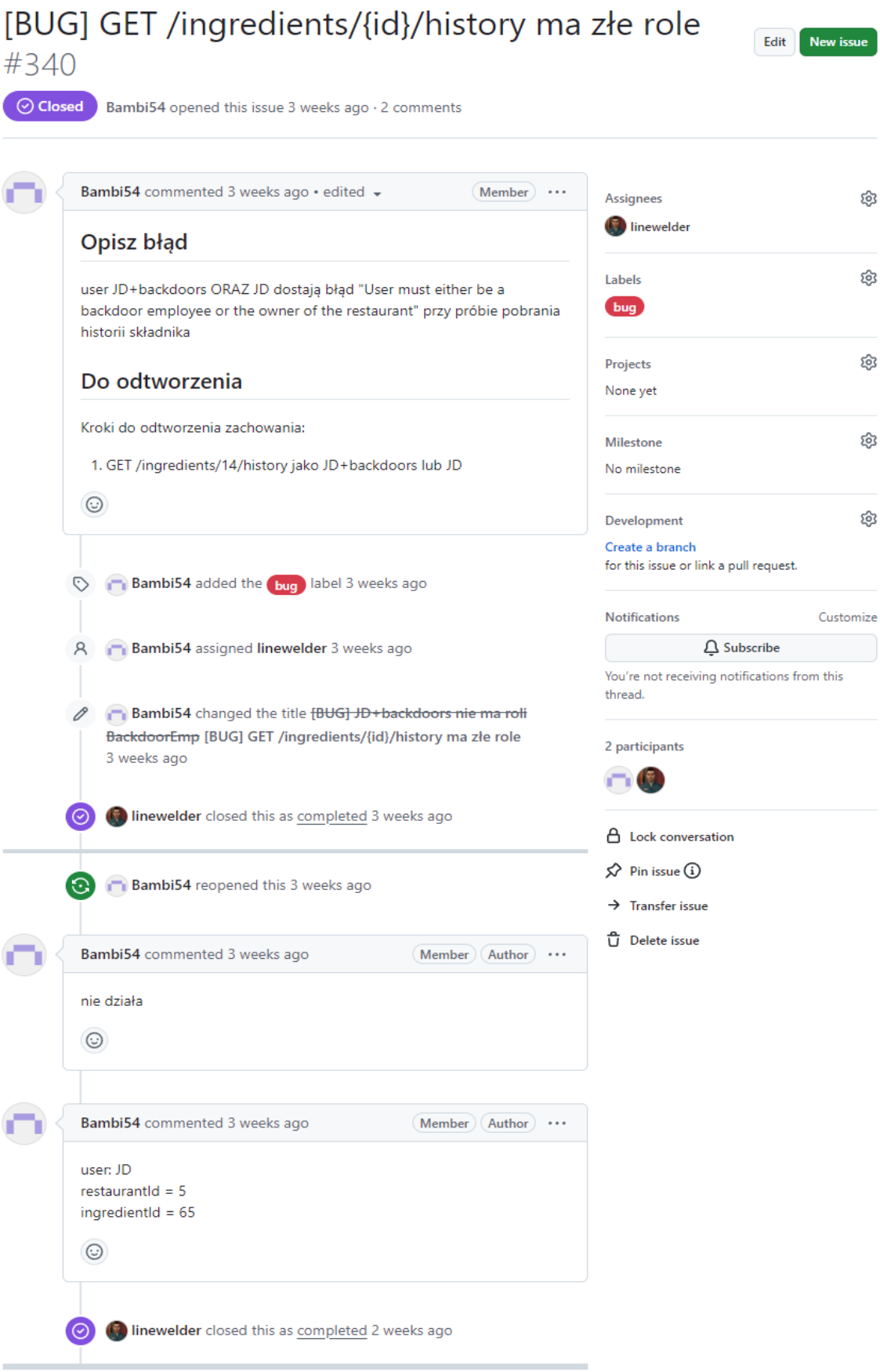
Kolejnym problemem było zwracanie użytkownikom końcowym błędów w formacie kodu pochodzących z metod serwisowych. Dla użytkownika aplikacji nie będącego deweloperem, kody te byłyby niezrozumiałe i mogłyby prowadzić do potencjalnej frustracji. W rozwiązaniu stworzono klasę "*Result*" realizującą wzorzec projektowy o tej samej nazwie. Klasa pozwala na zwróceniu wartości określonego typu, z metody.

W celu organizacji zgłaszanych błędów posłużono się platformą github, która umożliwia tworzenie tzw. "*Issues*" wraz z podziałem na kategorie (Rysunek 157). Zgłaszając napotkany problem należało wybrać kategorię, do której dany problem się klasyfikuje, co stanowiło dodatkową informację dla zespołu implementacji



Rysunek 157 Podział zgłoszeń na kategorie. Źródło: Opracowanie własne

Rysunek 158 przedstawia przykład jednego z wielu zgłoszeń jakie służyło do obsługiwania napotkanych problemów. Zazwyczaj takie zgłoszenia rozwiązuje się w języku angielskim, jednak ze względu na to, że wszyscy w zespole porozumiewali się w języku polskim zdecydowano się na używanie tego właśnie języka.



Rysunek 158 Przykładowa treść zgłoszenia. Źródło: Opracowanie własne

8.2.3 Frontend

Tak jak to miało miejsce przy implementacji backendu, dla frontendu również można było określić kategorię zgłaszanego problemu poprzez wykorzystanie niestandardowego modelu raportowania błędów na platformie Github.

W zespole frontendu pojawiły się liczne problemy organizacyjne i komunikacyjne, szczególnie na początkowym etapie prac. Zadania umieszczane na tablicy Trello były często zbyt ogólnikowe, co utrudniało ich realizację i prowadziło do nieporozumień. Dodatkowo członkowie zespołu nie zawsze pamiętali o przenoszeniu kart do odpowiednich kolumn, co powodowało chaos w śledzeniu postępów projektu. Problemy z organizacją pracy negatywnie wpłynęły na harmonogram, który wielokrotnie musiał być przesuwany, głównie z powodu trudności wynikających z niedostatecznych umiejętności zespołu w pisaniu kodu w TypeScript.

Brak doświadczenia w pracy z GitHubem w dużych projektach również okazał się istotnym wyzwaniem. Zarządzanie branchami i rozwiązywanie konfliktów podczas mergowania kodu z wielu źródeł często prowadziło do problemów, które wymagały dodatkowego czasu na naprawę. Kolejną trudnością było wybranie bibliotek do stylowania – Material UI i Tailwind CSS. Te dwie technologie, choć popularne, nie współpracują ze sobą dobrze, co skutkowało licznymi problemami z uzyskaniem spójnych i estetycznych stylów. W rezultacie część prac stylistycznych trzeba było wykonywać ręcznie, co znacznie opóźniało tempo realizacji projektu.

Należy również podkreślić, że żaden z członków zespołu nie miał dużego doświadczenia w szeroko pojętym frontendzie. Wiele kluczowych umiejętności, takich jak zarządzanie strukturą projektu czy zaawansowane korzystanie z frameworków, było zdobywane w trakcie prac. Taka sytuacja wymusiła wielokrotną refaktoryzację kodu, w tym istotnych elementów projektu, takich jak routing. W efekcie, choć zespół zdobył cenne doświadczenie i znacząco poszerzył swoje kompetencje, problemy organizacyjne, techniczne i braki w wiedzy znacząco wpłynęły na tempo oraz efektywność realizacji projektu.

8.2.4 Aplikacja mobilna

W zespole aplikacji mobilnej oprócz ogólnych problemów występujących dla całego projektu pojawiły się problemy związane z implementacją aplikacji. W rozwiązywaniu tego typu napotkanych trudności, pomocna okazała się dokumentacja aplikacji “Android studio” jak i strona internetowa “Stackoverflow”.

Jednym z napotkanych problemów występującym przy użyciu zdjęć w jednym z komponentów, była utrata niektórych atrybutów tj. obcięcie rogów czy zniekształcenie zdjęcia. Po żmudnych próbach naprawy wspomnianego błędu posiłkując się dokumentacją aplikacji Android w celu znalezienia rozwiązania, zdecydowano się na uaktualnienie wersji frameworku do wyższej. Okazało się, że w nowszej wersji przedstawiony błąd już nie występuje i dzięki temu aplikacja powróciła do poprawnego działania.

Kolejną trudnością, którą napotkał zespół implementacji aplikacji mobilnej było wyświetlanie użytkownikom, błędów zwracanych bezpośrednio przez serwer. Wyświetlanie użytkownikom informacji prosto z serwera było niepożądanym zachowaniem ze względu na roboczą treść takich błędów w języku angielskim. W celu naprawy, kody błędów zostały zmapowane do tzw. “*Resource*”, czyli napisów zawierających tłumaczenie. Pozwoliło to na kontrolowanie tekstu wyświetlanego użytkownikowi końcowemu z dodatkowym automatycznym tłumaczeniem.

8.2.5 Testy

Jedną z trudności, z którą spotkał się zespół testów była ograniczona możliwość współpracy nad testami końcówek backend, w aplikacji Postman, w wersji podstawowej nad jednym projektem z

testami mogły współpracować maksymalnie 3 osoby, przy czym zespół zajmujący się testami był większy. Dodatkowo aplikacja Postman nie pozwalała na zarządzanie wersjami przez Github.

W odpowiedzi na zaistniały problem zastosowana została stosunkowo nowa na rynku aplikacja Bruno API. Aspektami wyróżniającymi Bruno na tle Postmana, jest to, że w wersji podstawowej jest to aplikacja open source, co oznacza, że tym razem nad testami mogły współpracować wszystkie osoby z zespołu testowego, a do zarządzania wersjami użyty został Github. Dodatkowo Bruno API to aplikacja prosta i przyjemna w użytku co dodatkowo usprawniło pracę zespołu.

8.3 Dalszy rozwój

W omawianym projekcie wypełniono większość z założonych początkowo celów. Jednak ze względu na ograniczenia zasobów ludzkich (zarówno pod względem ilości osób jak i doświadczenia przy pracy zespołowej) oraz stanowczy limit czasowy, w końcowym systemie zabrakło funkcji, które uznano za przydatne, ale nie wymagane. Zostały one odrzucone wskutek potencjalnych, dodatkowych komplikacji implementacyjnych jak i projektowych. Do tych funkcjonalności należą:

- **Rozwinięcie księgowości i wystawianie faktur** - w obecnym stanie funkcje związane z przychodami oraz wydatkami restauracji ograniczają się do wyświetlających się w odpowiednim panelu statystyk. Uznano, że przydatne byłoby narzędzie pozwalające ułatwić np. rozliczanie podatków lub przeprowadzania inwentaryzacji;

- **Wdrożenie nawigacji użytkownika do lokalu** - obecna wersja systemu ogranicza się do przekazywania lokalizacji różnych restauracji oraz użytkownika. Brakuje jednak integracji systemu z nowoczesnymi metodami nawigacji, które mogłyby uwzględnić aktualny ruch, możliwe miejsca postojowe lub nawet połączenia transportu publicznego. Taka funkcja mogłaby być dostępna w formie jednego przycisku “nawiguj”, który mógłby się wyświetlać przy szczegółach restauracji;

- **Zarządzanie pensjami pracowników** – system obecnie nie obsługuje modułu do zarządzania wynagrodzeniami. W przyszłości można by rozbudować platformę o funkcjonalności obliczania wypłat w oparciu o godziny pracy lub zintegrować ją z systemami kadrowymi, co uprościłoby zarządzanie płatnościami dla właścicieli restauracji. Dodatkowo, taka funkcjonalność mogłaby obsługiwać różne rodzaje umów (np. umowa o pracę lub umowa zlecenie);

- **Rozwinięcie zarządzania magazynem** - aktualnie zarządzanie magazynem ogranicza się do prostego monitorowania stanu zapasów oraz śledzenia dokonanych zamówień u dostawców. Niestety zabrakło miejsca dla bardziej zaawansowanego narzędzia, które pozwoliłoby na integrację z systemami różnych dostawców lub chociażby możliwości śledzenia ułożenia asortymentu w magazynie;

- **Integracja z zewnętrznymi systemami dostaw** - system nie wspiera obecnie integracji z platformami zewnętrznymi umożliwiającymi dostarczanie posiłku do klientów. Aktualnie właściciel restauracji ma możliwość jedynie zaznaczyć w systemie czy wspiera dostawy czy też nie. Na rynku znajduje się wiele firm, które zajmują się dostawą jedzenia wyręczając przy tym restauracje. Spora część tych firm oferuje dostęp do swojego API właśnie w celu ułatwienia integracji z różnymi systemami POS;

- **Subskrypcje dla właścicieli restauracji** - system wyróżnia się tym, że w swoich założeniach jest darmowy i ogólnodostępny, można by jednak rozważyć wdrożenie modelu subskrypcyjnego. W ramach takiego modelu właściciele restauracji płaciliby miesięczną opłatę za dostęp do poszczególnych funkcji lub sposobów promowania wśród klientów. Model ten umożliwiłby także wprowadzenie różnych poziomów subskrypcji, dostosowanych do potrzeb restauracji różnej wielkości, co zwiększyłoby elastyczność systemu oraz jego rentowność;

8.4 Wnioski końcowe

Na przestrzeni trzech semestrów pracy nad projektem udało się zrealizować wszystkie założone cele. Efektem tych działań jest aplikacja Reservant, która została wyposażona w liczne funkcjonalności wyróżniające ją na tle konkurencyjnych systemów. Jednym z przykładów takich funkcji jest moduł kontroli stanu magazynu, umożliwiający właścicielowi lub managerowi restauracji na zdalne monitorowanie zasobów.

Kolejnym przykładem który aplikacja oferuje są funkcje społecznościowe, takie jak możliwość dodawania wydarzeń związanych z konkretnymi restauracjami czy komunikację poprzez czat tekstowy.

System został zaprojektowany z myślą o obsłudze różnych typów lokali gastronomicznych, takich jak restauracje, kawiarnie, fast foody czy bary. Dzięki temu każde z tych miejsc znajdzie swoje zastosowanie w aplikacji Reservant, co zapewnia użytkownikom dużą różnorodność dostępnych możliwości. Warto również zaznaczyć, że aplikacja posiada duży potencjał do dalszego rozwoju w przyszłości, zarówno w branży gastronomicznej jak i w innych obszarach wykraczających poza tę kategorię.

Niespełna półtoraroczna wspólna praca przyniosła każdemu członkowi zespołu wiele cennych wniosków i nowych umiejętności. Jednak najważniejszym aspektem tego procesu jest bez wątpienia doświadczenie zdobyte poprzez realizację różnorodnych zadań i wyzwań, przed którymi stanęli wszyscy zaangażowani w projekt studenci.

9 Bibliografia

[1]. Strona internetowa - POSBistro - Dostęp 19.09.2024
https://posbistro.com/pl/funkcje

[2]. Strona internetowa – Escasa – Dostęp 19.09.2024
https://www.escsa.pl/x2system-oprogramowanie-dla-gastronomii-i-hoteli

[3]. Strona internetowa - X2manager- Dostęp 19.09.2024
https://intentsales.pl/system-dla-gastronomii-x2system/

[4]. Strona internetowa - X2System- Dostęp 19.09.2024
https://www.escsa.pl/x2system-x2kasa-galeria

[5]. Strona internetowa - izzyRest - Dostęp 19.09.2024
https://segal.pl/program-dla-restauracji-4rest

[6]. Strona internetowa - izzyRest - Dostęp 19.09.2024
https://www.promokas.pl/p129,izzyrest-pos.html

[7]. Strona internetowa - FOODSOFT - Dostęp 19.09.2024
https://foodsoft.pl/oprogramowanie/

[8]. Strona internetowa - FOODSOFT - Dostęp 19.09.2024
https://foodsoft.pl/wersja-5-4/

[9]. Strona internetowa - FOODSOFT - Dostęp 19.09.2024
https://foodsoft.pl/wersja-5-3/

[10]. Strona internetowa - MojStolik - Dostęp 19.09.2024
https://www.mojstolik.pl/o-nas

[11]. Strona internetowa - MojStolik - Dostęp 19.09.2024
https://www.mojstolik.pl/dla-restauracji

[12]. Sklep Google Play - MojStolik - Dostęp 19.09.2024
https://play.google.com/store/apps/details?id=pl.mojstolik.restaurant&hl=pl

[13]. Strona internetowa - Zjedz.my - Dostęp 19.09.2024
https://zjedz.my/dla-restauracji

[14]. Strona internetowa - Zjedz.my - Dostęp 19.09.2024
https://zjedz.my/blog/zjedz-za-pol

[15]. Sklep Google Play – Zjedz.my - Dostęp 19.09.2024
https://play.google.com/store/apps/details?id=my.zjedz.android.company&pli=1

[16]. Strona internetowa - Dineout - Dostęp 19.09.2024
https://restaurant.dineout.pl/

[17]. Sklep Google Play – Dineout - Dostęp 20.09.2024
https://play.google.com/store/apps/details?id=com.bsbm.dineout&hl=pl

[18]. Strona internetowa - Antyweb - Dostęp 28.09.2024
https://antyweb.pl/dineout-poland-rezerwacja-stolikow-online-lub-w-aplikacji

[19]. Strona internetowa - Bookero - Dostęp 29.10.2024
https://www.bookero.pl/

[20]. Strona internetowa - Codearmour – Dostęp 05.11.2024
https://coderarmour.com/czym-jest-c/

[21]. Dokumentacja – Microsoft ASP .NET - Dostęp 05.11.2024
https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-8.0&tabs=visual-studio

[22]. Artykuł – Plukasiewicz – Dostęp 05.11.2024
https://www.plukasiewicz.net/Artykuly/AspNetCoreFeatures

[23]. Dokumentacja – Microsoft Entity Framework – Dostęp 05.11.2024
https://learn.microsoft.com/en-us/ef/core/dbcontext-configuration/

[24]. Artykuł - Plukasiewicz – Dostęp 05.11.2024
https://www.plukasiewicz.net/EFCore/Introduction

[25]. Artykuł – Medium – Dostęp 05.11.2024
https://medium.com/codenx/code-with-fluent-validations-in-net-c-da2fb517566d

[26]. Dokumentacja – FluentValidation – Dostęp 05.11.2024
https://docs.fluentvalidation.net/en/latest/aspnet.html?highlight=personvalidator

[27]. Strona internetowa – Wikipedia - Dostęp 05.11.2024
https://pl.wikipedia.org/wiki/JavaScript

[28]. Dokumentacja – React.js – Dostęp: 09.11.2024
https://pl.legacy.reactjs.org/docs/components-and-props.html

[29]. Strona internetowa - DigitalOcean – Dostęp: 11.01.2025
https://www.digitalocean.com/community/tutorials/how-to-create-custom-types-in-typescript

[30]. Dokumentacja – TypeScript – Dostęp: 09.11.2024
https://www.typescriptlang.org/

[31]. Dokumentacja – Material UI – Dostęp: 09.11.2024
https://mui.com/material-ui/

[32]. Dokumentacja – Material UI – Dostęp: 09.11.2024
https://mui.com/

[33]. Artykuł - Testim – Dostęp 05.11.2024
https://www.testim.io/blog/jest-testing-a-helpful-introductory-tutorial/

[34]. Dokumentacja – Jest – Dostęp: 09.11.2024
https://jestjs.io/

[35]. Dokumentacja – Jest – Dostęp: 09.11.2024
https://jestjs.io/docs/tutorial-react

[36]. Dokumentacja – Formik – Dostęp: 18.11.2024
https://formik.org/

[37]. Strona internetowa – CodeSandbox – Dostęp: 18.11.2024
https://codesandbox.io/p/sandbox/dazzling-swanson-wne32?file=%2Fsrc%2Findex.js%3A10%2C7-22%2C16&from-embed

[38]. Artykuł – Bring owl – Dostęp: 18.11.2024
https://boringowl.io/blog/dlaczego-warto-uzywac-yup-w-react

[39]. Dokumentacja - Yup – Dostęp: 18.11.2024
https://github.com/jquense/yup

[40]. Artykuł – Medium – Dostęp 18.11.2024
https://medium.com/@paulallles/validation-with-yup-bb12565c1f9d

[41]. Artykuł – Nearshore – Dostęp 18.11.2024
https://www.nearshore-it.eu/pl/artykuly/kotlin/

[42]. Strona internetowa – Oreilly – Dostęp 18.11.2024
https://www.oreilly.com/library/view/kotlin-for-enterprise/9781788997270/ea4ec584-db64-4026-89a8-2086301eb9c5.xhtml

[43]. Artykuł - Bugfender – Dostęp: 18.11.2024
https://bugfender.com/blog/kotlin-vs-java/

[44]. Artykuł - Dobreprogramy.pl – Dostęp: 18.11.2024
https://www.dobreprogramy.pl/android-sdk,program,windows,6628628395079809

[45]. Strona internetowa - Android – Dostęp: 18.11.2024
https://developer.android.com/studio

[46]. Dokumentacja – Truth – Dostęp: 18.11.2024
https://truth.dev/

[47]. Artykuł – Oracle – Dostęp 18.11.2024
https://www.oracle.com/pl/cloud/cloud-native/container-registry/what-is-docker/

[48]. Strona internetowa – Portainer – Dostęp: 18.11.2024
https://www.portainer.io/

[49]. Artykuł – SzkolaJenkinsa – Dostęp: 18.11.2024
https://szkolajenkinsa.pl/2021/02/07/czym-jest-jenkins/

[50]. Artykuł – Expose – Dostęp 18.11.2024
https://expose.pl/czym-jest-git-i-dlaczego-warto-znac-systemy-kontroli-wersji-kodu/

[51]. Artykuł – Droptica – Dostęp: 18.11.2024
https://www.droptica.pl/blog/co-jest-postman-do-czego-sluzy-i-jakie-sa-jego-funkcjonalnosci/

[52]. Artykuł – Medium – Dostęp: 27.11.2024
https://medium.com/@karthik9629846380/bruno-exploring-and-testing-apis-4d0e2bcd1604

[53]. Artykuł – Epic Games – Dostęp: 27.11.2024
https://store.epicgames.com/pl/news/what-is-discord-and-what-is-it-used-for

[54]. Artykuł – Artur Kosiński – Dostęp: 27.11.2024
https://arturkosinski.pl/baza-wiedzy/instrukcja-trello-proste-zespolowe-zarzadzanie-projektami-online/

[55]. Artykuł – Semcore – Dostęp: 27.11.2024
https://semcore.pl/co-to-jest-narzedzie-figma-do-czego-sluzy-i-na-jakich-etapach-projektowych-jest-uzyteczna/

[56]. Dokumentacja – IBM – Dostęp: 27.11.2024:
https://www.ibm.com/docs/pl/integration-bus/10.0?topic=apis-swagger

[57]. Artykuł – Testuj – Dostęp: 27.11.2024
https://testuj.pl/blog/wszystko-o-automatyzacji-testow-w-selenium/

[58]. Dokumentacja – Selenium – Dostęp: 27.11.2024
https://selenium-python.readthedocs.io/getting-started.html

[59]. Strona internetowa – Uptime Kuma – Dostęp: 11.01.2025
https://uptimekuma.eu

[60]. Repozytorium – Uptime Kuma – Dostęp: 11.01.2025
https://github.com/louislam/uptime-kuma

[61]. Strona internetowa – SmartFormat – Dostęp: 11.01.2025
https://github.com/axuno/SmartFormat

[62]. Entity Framework Core in Action 2 edycja - Jon P Smith, ISBN- 10: 1617298360, rok wydania: 2021, wydawnictwo: MANNING, rozdział 5

[63]. Repozytorium – Reservant – Dostęp: 21.12.2024
https://github.com/orgs/Reservant-inc/repositories

[64]. Strona internetowa - Infinum – Dostęp: 21.12.2024
https://infinum.com/blog/mvvm-architecture/

10 Spis ilustracji

11 Spis tabel

12 Spis listingów