



UAIM

Projekt - System rezerwacji hoteli

Łukasz Borowski (331159)

Mikołaj Brewczak (331160)

Kamil Gromko (331170)

Mateusz Kowalczuk (331175)

8 czerwca 2025

Spis treści

1. Cel projektu	3
2. Opis wykorzystanych technologii	3
3. Opis realizacji funkcjonalności	3
4. Backend	4
4.1. Architektura aplikacji	4
4.2. Struktura projektu	5
4.3. Modele danych	5
4.4. Endpointy API	5
4.5. Zabezpieczenia (dostęp do endpointów, przechowywanie haseł)	6
4.6. Wyniki testów jednostkowych (raport pokrycia kodu testami)	6
5. Opis aplikacji webowej	7
5.1. Architektura aplikacji	7
5.2. Struktura projektu	7
5.3. Widoki i powiązane z nimi funkcje aplikacji	8
5.4. Wyniki testów jednostkowych	8
Raport pokrycia kodu testami	9
6. Opis aplikacji mobilnej	9
6.1. Architektura aplikacji	9
6.2. Lista wykorzystywanych bibliotek zewnętrznych	10
6.3. Projekt widoków	10
6.4. Dokumentacja klas	11
7. Opis pliku konfiguracyjnego Docker	11
7.1. Definicje Usług (Services)	11
7.2. Sieci i Woluminy	12
8. Przedstawienie funkcjonalności aplikacji	13
8.1. Aplikacja mobilna	13
8.2. Aplikacja webowa	16
9. Wnioski	20

1. Cel projektu

Celem projektu jest implementacja i wdrożenie aplikacji internetowej oraz mobilnej, która służy do rezerwacji miejsc hotelowych. System jest rozwijany w środowisku konteneryzacji Docker, co ma naśladować warunki komercyjne. Aplikacja została przygotowana w sposób jak najbardziej zbliżony do uruchomienia komercyjnego.

Projekt zakłada budowę kompletnego systemu składającego się z warstwy backendowej odpowiedzialnej za logikę biznesową oraz obsługę danych, interfejsu użytkownika dostępnego z poziomu przeglądarki internetowej oraz aplikacji mobilnej, a także relacyjnej bazy danych, w której przechowywane są wszystkie informacje związane z użytkownikami, hotelami, pokojami oraz rezerwacjami. W ramach realizacji zaimplementowano również system uwierzytelniania oparty na tokenach JWT oraz wysyłkę potwierdzeń e-mail po wykonaniu rezerwacji lub jej anulowaniu, wraz z załączonym dokumentem (paragon lub faktura).

Aplikacja została zaprojektowana z myślą o użytkowniku końcowym, który w prosty i przejrzysty sposób może przeglądać oferty hoteli dostępnych w wybranej lokalizacji i terminie, a następnie zarezerwować wybrany pokój lub odwołać istniejącą rezerwację. Całość została przetestowana i przygotowana do pracy w środowisku uruchamianym za pomocą kontenerów Docker, co znacząco upraszcza wdrożenie na serwerze produkcyjnym oraz spełnia wymagania stawiane przez prowadzących w ramach przedmiotu.

Projekt ten nie tylko realizuje wymogi formalne, ale również odzwierciedla rzeczywiste podejście do tworzenia systemów webowych i mobilnych, z naciskiem na niezawodność, skalowalność oraz zgodność z dobrymi praktykami inżynierii oprogramowania.

2. Opis wykorzystanych technologii

System opiera się na architekturze składającej się z aplikacji front-endowej, back-endowej oraz bazy danych, z których każda działa w osobnym kontenerze Docker.

- **Front-End (Klient):** Aplikacja jednostronowa (SPA) stworzona w technologii React, działająca w przeglądarce internetowej użytkownika. Komunikuje się z back-endem poprzez wywołania API.
- **Back-End (Serwer):** Usługa odpowiedzialna za logikę biznesową aplikacji, napisana w języku Python z wykorzystaniem frameworka Flask. Udostępnia API dla aplikacji klienckich i zarządza komunikacją z bazą danych.
- **Baza Danych:** Relacyjna baza danych PostgreSQL przechowująca dane, działająca w kontenerze.
- **Aplikacja Mobilna:** Natywna aplikacja na system Android, napisana w języku Java, która komunikuje się z tym samym back-endem.
- **Serwer NGINX:** Serwer udostępniający zawartość statyczną, w przypadku aplikacji są to zdjęcia hoteli do wyświetlania we front-endzie i aplikacji mobilnej

3. Opis realizacji funkcjonalności

Poniższa tabela przedstawia listę funkcji wymaganych w ramach projektu portalu do rezerwacji miejsc hotelowych.

Lp.	Funkcjonalność	Status	Sposób realizacji
1	Zakładanie konta użytkownika (Rejestracja)	Zrealizowano	Formularz rejestracyjny w aplikacjach zbiera dane użytkownika, które następnie są przesyłane do API w celu utworzenia nowego użytkownika w bazie danych.
2	Uwierzytelnienie (Logowanie)	Zrealizowano	Formularz logowania weryfikuje dane użytkownika. Po pomyślnym uwierzytelnieniu generowany jest token JWT do zabezpieczenia dalszych żądań.
3	Przeglądanie ofert (daty, lokalizacja, udogodnienia, ceny)	Zrealizowano	Użytkownicy mogą przeglądać i filtrować oferty na podstawie lokalizacji (kraj i miasto, widok na mapie), dat (kalendarz), liczby osób, udogodnień. Wyświetlana jest również cena pobytu.
4	Dokonywanie rezerwacji dla podanej liczby osób	Zrealizowano	Użytkownicy mogą wybrać ofertę i zarezerwować pobyt dla określonej liczby osób w wybranym terminie, korzystając z interfejsu kalendarza.
5	Odwoływanie pobytu	Zrealizowano	Użytkownicy mają możliwość przeglądania swoich rezerwacji i ich anulowania. Anulowanie rezerwacji aktualizuje jej status w bazie danych.
6	Wysyłanie e-mailem potwierdzenia rezerwacji/odwołania razem z paragoldem/fakturą	Zrealizowano	Po dokonaniu lub odwołaniu rezerwacji, system automatycznie wysyła wiadomość e-mail z potwierdzeniem oraz odpowiednim dokumentem sprzedawy na adres użytkownika.

4. Backend

4.1. Architektura aplikacji

Aplikacja backendowa została zrealizowana w języku **Python** z wykorzystaniem lekkiego framework'a **Flask**. Jej architektura jest modularna i opiera się na wzorcu fabryki aplikacji (funkcja `create_app` w pliku `__init__.py`), co ułatwia zarządzanie konfiguracją i testowanie.

Kluczowe komponenty architektury to:

- **Flask-SQLAlchemy**: Służy jako warstwa ORM do interakcji z bazą danych PostgreSQL. Definicje modeli znajdują się w pliku `models.py`.
- **Flask-JWT-Extended**: Odpowiada za implementację uwierzytelniania przy użyciu tokenów JWT.
- **Blueprints**: Logika aplikacji została podzielona na moduły: `auth_bp` dla autoryzacji oraz `endp_bp` dla pozostałych endpointów aplikacji.
- **CORS (Cross-Origin Resource Sharing)**: Skonfigurowano mechanizm CORS, aby umożliwić komunikację z aplikacją frontendową działającą pod adresem `http://localhost:3000`.
- **Zarządzanie konfiguracją**: Wrażliwe dane, takie jak klucze sekretne, są ładowane ze zmiennych środowiskowych przy użyciu biblioteki `dotenv`, co zapobiega ich umieszczaniu bezpośrednio w kodzie.
- **Serwer NGINX**: NGINX odpowiada za serwowanie plików statycznych, takich jak zdjęcia hoteli. W konfiguracji serwera zmapowano dedykowany folder zawierający obrazy, dzięki czemu są one dostępne publicznie. Odwołania do tych zasobów (np. `http://localhost:8888/images/hotels/...`) zostały umieszczone w skrypcie inicjalizującym bazę danych `init.sql`.

4.2. Struktura projektu

Projekt ma logiczną strukturę katalogów, która oddziela od siebie poszczególne warstwy aplikacji:

Kod 1. Rzeczywista struktura projektu (na podstawie VS Code)

```
1 / (katalog główny projektu)
2 |
3 |-- /back-end
4 | |-- /flaskr
5 | | |-- __init__.py          # Inicjalizacja aplikacji Flask ( wzorzec fabryki )
6 | | |-- authorization.py    # Endpointy autoryzacji i rejestracji
7 | | |-- endpoints.py         # Główne endpointy API aplikacji
8 | | |-- models.py            # Modele danych SQLAlchemy
9 | | |-- validators.py        # Funkcje walidujące dane wejściowe
10 | | `-- extensions.py       # Inicjalizacja rozszerzeń (np. db)
11 |
12 | |-- /tests
13 | | |-- test_authorization.py # Testy jednostkowe dla autoryzacji
14 | | |-- test_endpoints.py     # Testy jednostkowe dla endpointów
15 | | |-- ...
16 | | |
17 | | |-- config.py           # Plik konfiguracyjny
18 | | |-- Dockerfile           # Instrukcje budowania kontenera dla back-endu
19 | | |-- mailer.py            # Moduł odpowiedzialny za wysyłkę e-maili
20 | | `-- requirements.txt      # Zależności środowiska Python
21 |
22 |-- /db
23 |   `-- init.sql            # Skrypt do tworzenia i zasilania bazy danych
24 |
25 |-- /image-server
26 |   |-- /images/             # Katalog ze zdjęciami hoteli
27 |   |-- Dockerfile           # Instrukcje budowania kontenera NGINX
28 |   `-- nginx.conf            # Plik konfiguracyjny serwera NGINX
```

4.3. Modele danych

Modele danych zostały zdefiniowane przy użyciu klas SQLAlchemy w pliku `models.py`. Struktura bazy danych jest również opisana w pliku `init.sql`, który tworzy tabele i wypełnia je danymi początkowymi.

Główne modele to:

- **User**: Przechowuje dane użytkowników, w tym zahaszowane hasło.
- **Address, Hotel, HotelImage**: Reprezentują hotele, ich adresy oraz powiązane zdjęcia.
- **Room**: Definiuje pokoje dostępne w hotelu.
- **Reservation**: Łączy użytkownika (**User**) i pokój (**Room**), przechowując szczegóły rezerwacji.
- **HotelFacility** i **RoomFacility**: Tabele słownikowe z udogodnieniami.
- **HotelHotelFacility** i **RoomRoomFacility**: Tabele pośredniczące implementujące relację wiele-do-wielu.

4.4. Endpointy API

Aplikacja udostępnia API typu REST, zaimplementowane w plikach `authorization.py` i `endpoints.py`.

Autoryzacja (`authorization.py`):

- `POST /register`: Rejestracja nowego użytkownika.
- `POST /login`: Logowanie i zwracanie tokenów JWT.

Wyszukiwanie i Rezerwacje (`endpoints.py`):

- `POST /search_free_rooms`: Zaawansowane wyszukiwanie dostępnych pokoi.
- `POST /post_reservation`: Tworzenie nowej rezerwacji (wymaga JWT).
- `POST /post_cancellation`: Anulowanie rezerwacji (wymaga JWT).

Zarządzanie użytkownikiem (`endpoints.py`, wymaga JWT):

- GET /user/<id_user>: Pobieranie danych o użytkowniku.
- PUT /user/<id_user>/password: Zmiana hasła.
- DELETE /user/<id_user>: Usunięcie konta.
- GET /user/<id_user>/reservations: Pobieranie rezerwacji użytkownika.

4.5. Zabezpieczenia (dostęp do endpointów, przechowywanie haseł)

W aplikacji zaimplementowano wielopoziomowe mechanizmy bezpieczeństwa:

- **Przechowywanie haseł:** Hasła są haszowane za pomocą `generate_password_hash` i weryfikowane przez `check_password_hash`.
- **Uwierzytelnianie i autoryzacja:** Dostęp do wrażliwych endpointów jest chroniony tokenami JWT i dekoratorem `@jwt_required()`. System weryfkuje tożsamość użytkownika (`get_jwt_identity()`), aby zapobiec dostępowi do cudzych danych.
- **Walidacja danych wejściowych:** Dane przychodzące od klienta są walidowane po stronie serwera pod kątem obecności wymaganych pól i poprawności formatu.
- **Bezpieczna konfiguracja:** Klucze `SECRET_KEY` i `JWT_SECRET_KEY` są ładowane ze zmiennych środowiskowych.

4.6. Wyniki testów jednostkowych (raport pokrycia kodu testami)

W celu zapewnienia niezawodności działania backendu aplikacji, przygotowany został zestaw ponad stu testów jednostkowych, które zostały napisane w języku Python z wykorzystaniem biblioteki `unittest`. Testy obejmują wszystkie kluczowe komponenty aplikacji, w tym moduły odpowiedzialne za:

- rejestrację i logowanie użytkownika (`authorization.py`),
- walidację danych wejściowych (`validators.py`),
- obsługę żądań API i logikę rezerwacyjną (`endpoints.py`),
- generowanie i wysyłkę wiadomości email (`mailer.py`),
- logikę wewnętrznych modeli danych i mechanizmy bezpieczeństwa haseł (`models.py`).

Każdy z powyższych komponentów posiada dedykowany zestaw testów (`test_*.py`), które uruchamiane są na odizolowanej bazie danych typu `SQLite in-memory`. Dzięki temu możliwa była dokładna weryfikacja poprawności logiki aplikacji w kontrolowanych warunkach, bez wpływu na dane produkcyjne.

Do pomiaru pokrycia kodu testami wykorzystano narzędzie `coverage.py`. Komenda `coverage report -m` wykazała następujące statystyki:

Name	Stmts	Miss	Branch	BrPart	Cover
config.py	11	0	0	0	100%
flaskr/__init__.py	59	18	6	2	66%
flaskr/authorization.py	48	5	14	0	92%
flaskr/endpoints.py	371	24	134	10	93%
flaskr/extensions.py	2	0	0	0	100%
flaskr/models.py	74	0	0	0	100%
flaskr/validators.py	23	0	10	0	100%
mailer.py	43	0	4	0	100%
tests/test_authorization.py	88	1	2	1	98%
tests/test_endpoints.py	740	2	36	8	99%
tests/test_mailer.py	41	1	2	1	95%
tests/test_models.py	28	1	2	1	93%
tests/test_validators.py	47	1	2	1	96%
TOTAL	1575	53	212	24	96%

Z raportu wynika, że całkowite pokrycie kodu testami wyniosło **96%**, co znacznie przekracza minimalny wymagany próg 60% określony w założeniach projektowych. W szczególności, kluczowe komponenty takie jak validatory, modele danych i obsługa maili osiągnęły pokrycie na poziomie 100%.

Testy dla endpointów REST API obejmują zarówno przypadki pozytywne (np. poprawna rezerwacja i anulowanie), jak i negatywne (brak uprawnień, nieprawidłowe dane, próba rezerwacji zajętego pokoju itp.). W testach uwzględniono także warstwę autoryzacyjną opartą na tokenach JWT.

Wysoki poziom pokrycia testami, w połączeniu z ich rozbudowaną strukturą i testowaniem przypadków brzegowych, pozwala uznać backend projektu za dobrze zabezpieczony przed typowymi błędami programistycznymi i gotowy do dalszego rozwoju oraz wdrożenia produkcyjnego.

5. Opis aplikacji webowej

Aplikacja webowa została zrealizowana jako **Single Page Application (SPA)** przy użyciu biblioteki **React**. Zapewnia ona użytkownikom interfejs do wyszukiwania, przeglądania i rezerwowania pokoi hotelowych, a także do zarządzania swoim kontem i rezerwacjami.

5.1. Architektura aplikacji

Architektura aplikacji jest nowoczesna i opiera się na sprawdzonych wzorcach stosowanych w ekosystemie React.

- **Struktura komponentowa:** Aplikacja jest zbudowana z reużywalnych komponentów. Wyróżnione komponenty-strony (folder `/pages`), które odpowiadają za całe widoki, oraz mniejsze, ogólne komponenty UI (np. `Button`, `Card`) i layoutu (`PageLayout`).
- **Routing po stronie klienta:** Nawigacja w aplikacji jest obsługiwana przez bibliotekę **React Router DOM**. Główny plik `App.js` definiuje wszystkie dostępne ścieżki (trasy), w tym publiczne, chronione (tylko dla zalogowanych) i “public-only” (tylko dla niezalogowanych).
- **Zarządzanie stanem globalnym:** Stan autoryzacji użytkownika (informacje o zalogowanym użytkowniku oraz tokeny) jest zarządzany globalnie przy użyciu **React Context API**. Plik `AuthContext.js` tworzy dostawcę (`AuthProvider`), który udostępnia dane użytkownika oraz funkcje `login`, `register` i `logout` w całej aplikacji.
- **Utrwalanie sesji:** Informacje o zalogowanym użytkowniku oraz token JWT są przechowywane w `localStorage` przeglądarki. Dzięki temu sesja użytkownika jest przywracana po odświeżeniu strony.
- **Komunikacja z API:** Wszystkie zapytania do backendu są realizowane poprzez skoncentrowany moduł API (plik ‘`api.js`’, którego istnienie jest wywnioskowane z kodu), który najprawdopodobniej jest prekonfigurowaną instancją klienta HTTP, np. Axios.
- **Ochrona ścieżek:** Dostęp do poszczególnych widoków jest kontrolowany przez komponent **ProtectedRoute**, który na podstawie stanu z `AuthContext` decyduje, czy użytkownik ma uprawnienia do wyświetlenia danej strony, czy powinien zostać przekierowany (np. do strony logowania).
- **Styling:** Aplikacja wykorzystuje hybrydowe podejście do stylizacji, łącząc globalne style i framework **Tailwind CSS** (skonfigurowany w `index.css`) z dedykowanymi plikami CSS dla poszczególnych komponentów (np. `Navbar.css`).

5.2. Struktura projektu

Struktura projektu jest oparta na standardowym szablonie tworzonym przez `create-react-app`, z logicznym podziałem na katalogi w zależności od przeznaczenia plików, co widać na poniższym schemacie.

Kod 2. Rzeczywista struktura projektu aplikacji webowej

```
1 / (katalog główny projektu)
2 |
3 |-- /src
4 |   |-- /api
```

```

5 |   |   `-- api.js                      # Scentralizowana konfiguracja klienta API (np. Axios)
6 |
7 |   |-- /components
8 |   |   |-- /layout
9 |   |   |   `-- PageLayout.js          # Komponent ogólnego layoutu (stopka, nagłówek)
10 |   |   |-- /ui
11 |   |   |   |-- Button.js            # Reuzywalny komponent przycisku
12 |   |   |   |-- Card.js              # Reuzywalny, stylizowany kontener
13 |   |   |   |-- Navbar.js            # Komponent paska nawigacyjnego
14 |   |   |   |-- ProtectedRoute.js    # Komponent do ochrony ścieżek
15 |   |   |   `-- RoomCard.js          # Karta wyświetlająca podsumowanie oferty pokoju
16 |
17 |   |-- /contexts
18 |   |   `-- AuthContext.js         # Kontekst Reacta do zarządzania stanem autoryzacji
19 |
20 |   |-- /pages
21 |   |   |-- LoginPage.js           # Widok strony logowania
22 |   |   |-- RegisterPage.js        # Widok strony rejestracji
23 |   |   |-- SearchPage.js          # Widok głównej wyszukiwarki
24 |   |   |-- RoomDetailPage.js      # Widok szczegółów oferty
25 |   |   |-- MyReservationsPage.js  # Widok rezerwacji użytkownika
26 |   |   |-- ProfilePage.js         # Widok profilu użytkownika
27 |   |   `-- NotFoundPage.js        # Widok strony 404
28 |
29 |   |-- App.js                    # Główny komponent aplikacji z routingu
30 |   |-- index.js                 # Punkt wejściowy aplikacji, renderuje App
31 |   `-- index.css                # Globalne style i konfiguracja Tailwind CSS
32 |
33 |-- package.json                # Definicje projektu i zależności bibliotek
34 |-- postcss.config.js
35 |-- tailwind.config.js

```

5.3. Widoki i powiązane z nimi funkcje aplikacji

Aplikacja składa się z kilku głównych widoków (stron), z których każdy realizuje określone zadania.

- **LoginPage.js:** Zawiera formularz logowania. Po poprawnym uwierzytelnieniu użytkownika za pomocą funkcji z `AuthContext`, następuje przekierowanie do głównego panelu wyszukiwania.
- **RegisterPage.js:** Udostępnia formularz rejestracyjny. Po pomyślnym przesłaniu danych i utworzeniu konta, użytkownik jest informowany o sukcesie i przekierowywany do strony logowania.
- **SearchPage.js:** Główny widok aplikacji, dostępny dla wszystkich użytkowników. Pozwala na wyszukiwanie pokoi za pomocą rozbudowanego formularza z wieloma filtrami (daty pobytu, liczba gości, cena, udogodnienia, lokalizacja, standard hotelu). Po wysłaniu zapytania do API, wyniki są wyświetlane jako lista komponentów `RoomCard`.
- **RoomDetailPage.js:** Widok szczegółów wybranego pokoju, dostępny dla wszystkich. Prezentuje dane o pokoju i hotelu, galerię zdjęć, a dla zalogowanych użytkowników – przycisk do dokonania rezerwacji, który otwiera modal z potwierdzeniem.
- **MyReservationsPage.js:** Strona dostępna tylko dla zalogowanych użytkowników. Pobiera z API i wyświetla listę aktywnych rezerwacji danego użytkownika, dając jednocześnie możliwość ich anulowania.
- **ProfilePage.js:** Chroniony widok profilu użytkownika. Wyświetla jego dane osobowe oraz udostępnia formularz do zmiany hasła i przycisk do trwałego usunięcia konta.
- **NotFoundPage.js:** Prosty widok informujący o błędzie 404, wyświetlany, gdy użytkownik próbuje uzyskać dostęp do nieistniejącej ścieżki w aplikacji.

5.4. Wyniki testów jednostkowych

Aplikacja frontendowa została poddana testom jednostkowym i integracyjnym przy użyciu frameworka **Jest** oraz biblioteki **React Testing Library**. Zgodnie z popularną praktyką, pliki testowe (`*.test.js`) zostały umieszczone w tych samych katalogach, co testowane przez nie komponenty i strony.

Celem testów było sprawdzenie poprawności renderowania komponentów, ich reakcji na interakcje użytkownika oraz logiki biznesowej.

Raport pokrycia kodu testami

Wygenerowany raport pokrycia kodu testami pokazuje, że aplikacja **znacząco przekracza projektowe wymaganie 60% pokrycia**. Całkowite wyniki dla projektu prezentują się następująco:

- **Instrukcje (Stmts): 84.9%**
- **Rozgałęzienia (Branch): 73.97%**
- **Funkcje (Funcs): 76.23%**
- **Linie (Lines): 89.7%**

File	% Stmt	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	84.9	73.97	76.23	89.7	
src	100	100	100	100	
App.js	100	100	100	100	
src/api	87.5	50	100	87.5	
api.js	87.5	50	100	87.5	11
src/components	100	82.92	100	100	
Navbar.js	100	71.42	100	100	32, 41-54
ProtectedRoute.js	100	88.88	100	100	5
RoomCard.js	100	88.88	100	100	27-28
src/components/layout	100	100	100	100	
PageLayout.js	100	100	100	100	
src/components/ui	100	100	100	100	
Button.js	100	100	100	100	
Card.js	100	100	100	100	
src/context	95.23	83.33	100	100	
AuthContext.js	95.23	83.33	100	100	14
src/pages	82.85	71.59	70.37	88.07	
LoginPage.js	100	100	100	100	
MyReservationsPage.js	97.14	91.66	100	97.05	34
NotFoundPage.js	100	100	100	100	
ProfilePage.js	95.65	93.75	88.88	97.72	155
RegisterPage.js	100	100	100	100	
RoomDetailPage.js	75.32	70	69.23	77.33	53, 56, 75-82, 93-95, 98-99, 102-103, 136, 144, 220-238
SearchPage.js	73.98	55.73	54.76	85	63, 66, 76, 123-127, 238-277, 296, 317-334, 350, 399

Rysunek 1. Wyniki testów pokrycia dla aplikacji front-end

Analiza szczegółowa raportu wskazuje, że kluczowe komponenty reużywalne (np. `Button.js`, `Card.js`), komponenty layoutu (`PageLayout.js`) oraz proste strony (`LoginPage.js`, `RegisterPage.js`) osiągnęły **100% pokrycia**, co świadczy o ich wysokiej niezawodności.

Nieco niższe pokrycie (choć wciąż wysokie i znacznie powyżej progu 60%) wykazują bardziej złożone komponenty, takie jak `SearchPage.js` (73.98% instrukcji) i `RoomDetailPage.js` (75.32% instrukcji). Wynika to z dużej liczby rozgałęzień logicznych i interakcji, a raport precyzyjnie wskazuje nieprzetestowane linie kodu, które mogą stać się celem dalszych prac nad ulepszeniem zestawu testów.

6. Opis aplikacji mobilnej

Aplikacja mobilna została stworzona natywnie na platformę Android w języku Java. Umożliwia użytkownikom wyszukiwanie, rezerwację i zarządzanie pobytami hotelowymi, komunikując się z przygotowanym wcześniej API backendowym.

6.1. Architektura aplikacji

Architektura aplikacji opiera się na klasycznym, wieloaktywnościowym (multi-activity) podejściu, gdzie każdy ekran aplikacji jest oddzielną klasą dziedziczącą po `AppCompatActivity`.

- **Przepływ danych:** Dane pomiędzy aktywnościami są przekazywane za pomocą obiektów Intent. Złożone obiekty danych, takie jak UserAndTokens, Room czy Reservation, implementują interfejs Parcelable, co umożliwia ich efektywną serializację i transport między ekranami.
- **Logika i UI:** Logika biznesowa, w tym obsługa interfejsu użytkownika oraz wykonywanie zapytań sieciowych, jest w dużej mierze zawarta bezpośrednio w klasach aktywności.
- **Komunikacja sieciowa:** Za całą komunikację z API RESTful odpowiada biblioteka OkHttp3. Zapytania wykonywane są asynchronicznie, a odpowiedzi i aktualizacje interfejsu użytkownika są realizowane w głównym wątku UI za pomocą metody runOnUiThread. Aplikacja jest skonfigurowana do komunikacji z lokalnym serwerem deweloperskim (adres 10.0.2.2) poprzez wpis w pliku network_security_config, zadeklarowany w manifeście.
- **Przetwarzanie JSON:** Do parsowania danych w formacie JSON wykorzystano różne metody: bibliotekę Jackson Databind (ObjectMapper) oraz standardowe klasy org.json.JSONObject. oraz biblioteki Jackson

6.2. Lista wykorzystywanych bibliotek zewnętrznych

- **OkHttp3:** Nowoczesny i wydajny klient HTTP do komunikacji z backendem.
- **Jackson Databind:** Biblioteka do łatwego mapowania danych JSON na obiekty Java (POJO).
- **Glide:** Zaawansowana biblioteka do ładowania, buforowania i wyświetlania obrazów z adresów URL.
- **Google Play Services (Maps):** Umożliwia integrację i wyświetlanie interaktywnych Map Google w aplikacji.
- **AndroidX Libraries:** Zestaw podstawowych bibliotek, w tym appcompat, recyclerview, viewpager2 oraz constraintlayout.
- **Google Material Components:** Biblioteka dostarczająca nowoczesne komponenty UI, takie jak przyciski, pola tekstowe, okna dialogowe i selektory dat.

6.3. Projekt widoków

Interfejs użytkownika został zaprojektowany w serii połączonych ze sobą ekranów, zdefiniowanych w plikach layoutu XML.

- **Przepływ w aplikacji:** Użytkownik rozpoczyna od ekranu logowania (LoginActivity), który jest punktem startowym aplikacji. Stamąd może zalogować się lub przejść do ekranu rejestracji (RegisterActivity). Po zalogowaniu trafia do MainActivity – głównego panelu wyszukiwania, gdzie może definiować kryteria i szukać ofert.
- **Ekrany i ich layouty:**
 - **Wyniki wyszukiwania:** SearchResultsActivity wyświetla listę pokoi w RecyclerView, korzystając z item_room.xml do zdefiniowania wyglądu pojedynczego elementu.
 - **Szczegóły oferty:** Po wybraniu pokoju użytkownik przechodzi do SingleOfferActivity, gdzie widzi szczegółowe informacje, galerię zdjęć (ViewPager2), mapę z lokalizacją hotelu oraz formularz rezerwacji.
 - **Konto użytkownika:** AccountDetails to ekran profilu, na którym wyświetlane są dane użytkownika i lista jego aktywnych rezerwacji (item_reservation.xml). Z tego miejsca można anulować rezerwację lub przejść do zmiany hasła w ChangePassword.
- **Stylistyka:** Widoki wykorzystują ConstraintLayout do budowy bardziej skomplikowanych interfejsów oraz LinearLayout do prostszych układów. CardView został użyty do nadania elementom list nowoczesnego wyglądu z cieniem i zaokrąglonymi rogami.

6.4. Dokumentacja klas

Wszystkie dostarczone klasy Javy zawierają komentarze dokumentacyjne wyjaśniające ich przeznaczenie.

- **Aktywności (Activities):**
 - **LoginActivity:** Obsługuje logikę ekranu logowania, jest punktem wejścia do aplikacji.
 - **RegisterActivity:** Zarządza formularzem i procesem rejestracji nowego użytkownika.
 - **MainActivity:** Główny ekran po zalogowaniu, pełni rolę panelu do wyszukiwania ofert hotelowych z licznymi filtrami.
 - **SearchResultsActivity:** Odpowiada za wyświetlenie wyników wyszukiwania pobranych z API.
 - **SingleOfferActivity:** Prezentuje szczegółowe dane pojedynczej oferty i obsługuje proces rezerwacji.
 - **AccountDetails:** Ekran profilu użytkownika z listą rezerwacji i opcją zmiany hasła.
 - **ChangePassword:** Formularz pozwalający użytkownikowi na zmianę swojego hasła.
- **Modele danych (POJO):**
 - **UserAndTokens:** Klasa przechowująca dane zalogowanego użytkownika oraz jego tokeny dostępowe JWT. Implementuje **Parcelable**.
 - **Room:** Reprezentuje pojedynczą ofertę pokoju hotelowego. Implementuje **Parcelable**.
 - **Reservation:** Przechowuje wszystkie dane dotyczące jednej rezerwacji. Implementuje **Parcelable**.
- **Adaptery (Adapters):**
 - **RoomAdapter:** Adapter dla **RecyclerView**, który wiąże listę obiektów **Room** z widokami **item_room.xml**.
 - **ReservationAdapter:** Adapter dla **RecyclerView**, który wyświetla listę rezerwacji, używając widoku **item_reservation.xml**.
 - **ImagePagerAdapter:** Adapter dla **ViewPager2**, który tworzy przesuwaną galerię zdjęć.

7. Opis pliku konfiguracyjnego Docker

Cały system, z wyjątkiem natywnej aplikacji mobilnej, jest zarządzany i uruchamiany za pomocą narzędzia **Docker Compose**. Konfiguracja została zdefiniowana w pliku **docker-compose.yml**, który orkiestruje cztery główne, połączone ze sobą usługi: bazę danych, aplikację backendową, serwer NGINX do plików statycznych oraz aplikację frontendową.

7.1. Definicje Usług (Services)

Plik definiuje następujące kontenery:

- **db (Baza Danych)**
 - **Obraz:** Usługa bazuje na oficjalnym obrazie **postgres:latest**.
 - **Konfiguracja:** Za pomocą zmiennych środowiskowych (**POSTGRES_USER**, **POSTGRES_PASSWORD**, **POSTGRES_DB**) tworzona jest baza danych **hotel_db** wraz z danymi dostępowymi.
 - **Woluminy:** Zastosowano dwa woluminy:
 1. Nazwany wolumin **postgres_data** jest mapowany do ścieżki **/var/lib/postgresql/data** w kontenerze, co zapewnia trwałość (persistencję) danych bazy nawet po zatrzymaniu lub usunięciu kontenera.

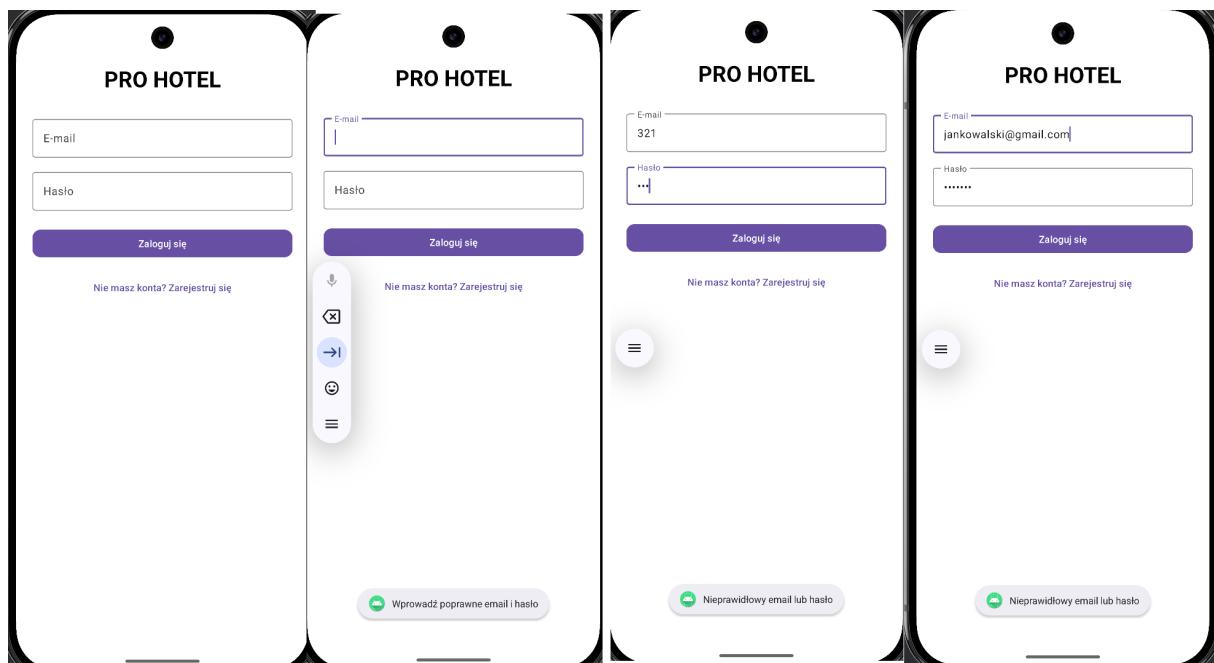
2. Plik `./db/init.sql` jest montowany w `/docker-entrypoint-initdb.d/init.sql`. Obraz `postgres` automatycznie wykonuje skrypty z tego katalogu przy pierwszym uruchomieniu, co pozwala na inicjalizację schematu bazy danych i wgranie danych początkowych.
- **Sieć:** Usługa jest podłączona do wspólnej sieci `hotel_network`.
 - **backend (Aplikacja Backendowa)**
 - **Budowanie:** Kontener jest budowany na podstawie `Dockerfile` znajdującego się w katalogu `./back-end`.
 - **Uruchomienie:** Polecenie `flask run -host=0.0.0.0` uruchamia serwer deweloperski Flask, który nasłuchuje na wszystkich interfejsach sieciowych wewnątrz kontenera.
 - **Zależności:** Usługa `backend` ma zdefiniowaną zależność (`depends_on`) od usługi `db`, co gwarantuje, że kontener z bazą danych zostanie uruchomiony przed kontenerem z aplikacją backendową.
 - **Sieć:** Usługa jest podłączona do sieci `hotel_network`, co umożliwia jej komunikację z bazą danych poprzez nazwę usługi (`db`).
 - **nginx (Serwer Plików Statycznych)**
 - **Obraz:** Usługa korzysta z lekkiego obrazu `nginx:alpine`.
 - **Rola:** Jej głównym zadaniem jest serwowanie statycznych plików (zdjęć hoteli). Nie pełni roli reverse proxy dla całej aplikacji, lecz dedykowanego serwera obrazów.
 - **Porty:** Port 80 kontenera jest zmapowany na port 8888 maszyny hosta.
 - **Woluminy:**
 1. Lokalny plik `./image-server/nginx.conf` jest montowany jako główny plik konfiguracyjny NGINX.
 2. Lokalny katalog `./image-server/images` jest montowany wewnątrz kontenera, dzięki czemu NGINX ma dostęp do zdjęć i może je serwować.
 - **frontend (Aplikacja Frontendowa)**
 - **Budowanie:** Kontener jest budowany z `Dockerfile` w katalogu `./front-end/terminal`.
 - **Środowisko:** Zmienna `CHOKIDAR_USEPOLLING=true` jest ustawiona w celu zapewnienia poprawnego działania mechanizmu hot-reloading serwera deweloperskiego React w środowisku Docker.
 - **Porty:** Standardowy port aplikacji React (3000) jest udostępniony na zewnątrz.
 - **Zależności:** Usługa jest zależna od `backend`, ponieważ do poprawnego działania potrzebuje dostępu do API.

7.2. Sieci i Woluminy

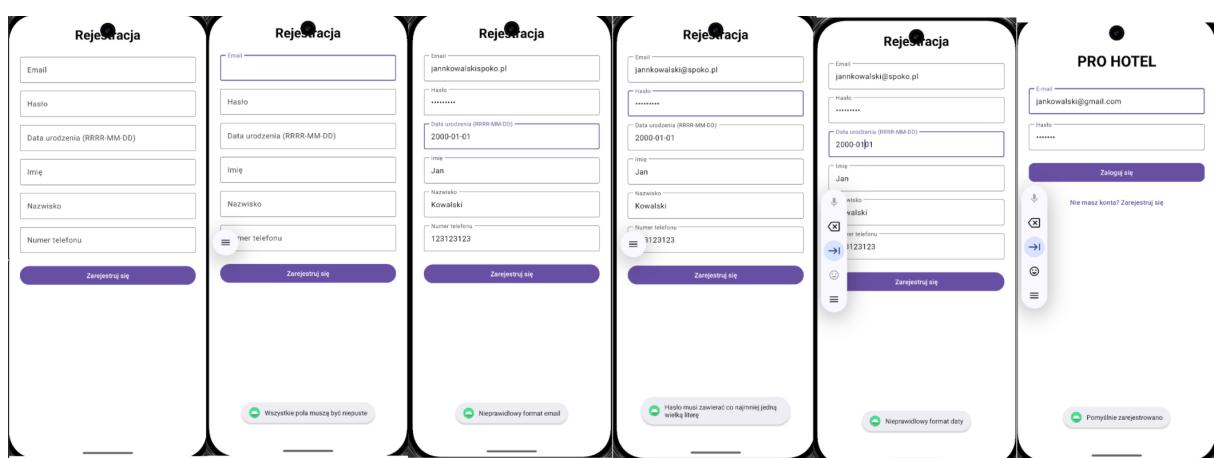
- **Sieci:** Zdefiniowano jedną, wspólną sieć typu `bridge` o nazwie `hotel_network`. Umożliwia ona płynną komunikację pomiędzy wszystkimi kontenerami za pomocą ich nazw (np. `backend` może odwołać się do bazy danych pod adresem `db:5432`).
- **Woluminy:** Zdefiniowano nazwany wolumin `postgres_data`, aby oddzielić dane bazy danych od cyklu życia kontenera `db` i zapobiec ich utracie.

8. Przedstawienie funkcjonalności aplikacji

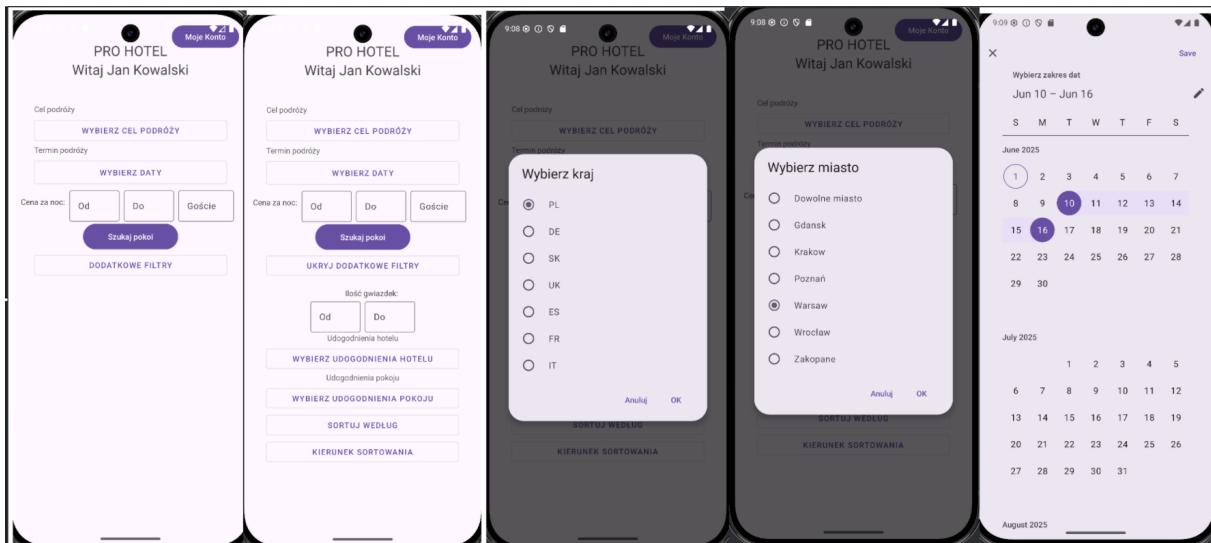
8.1. Aplikacja mobilna



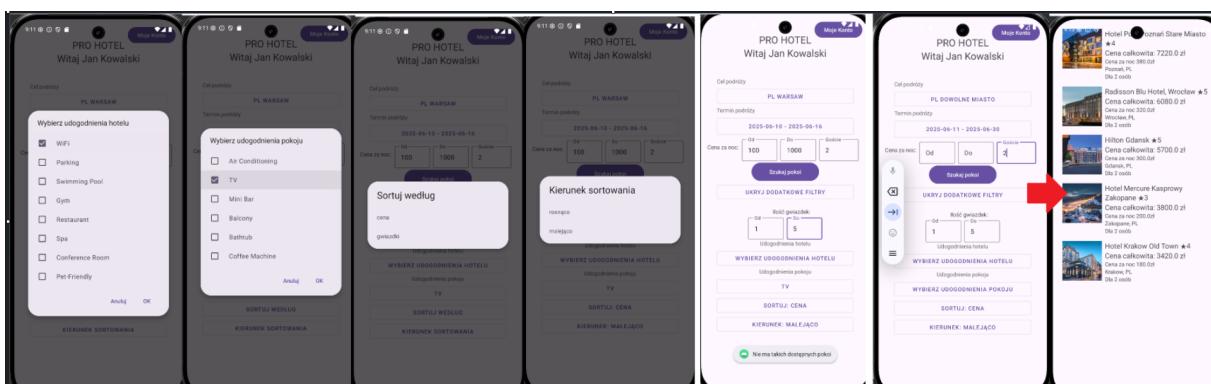
Rysunek 2. Widok logowania do aplikacji



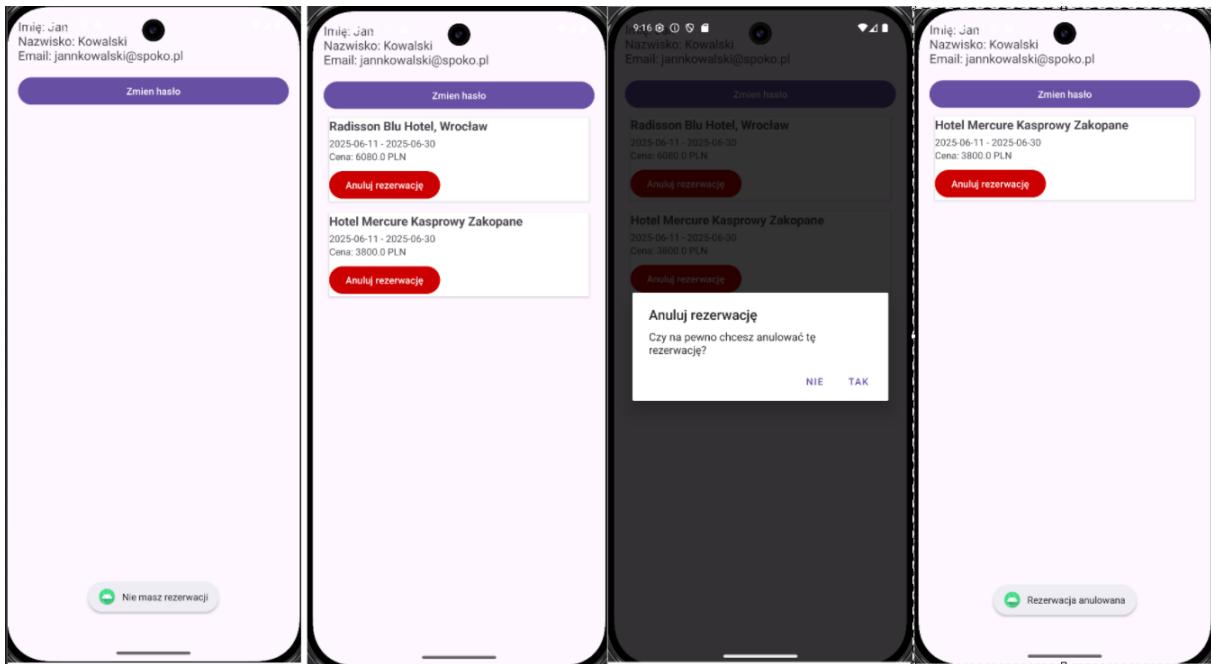
Rysunek 3. Widok rejestracji



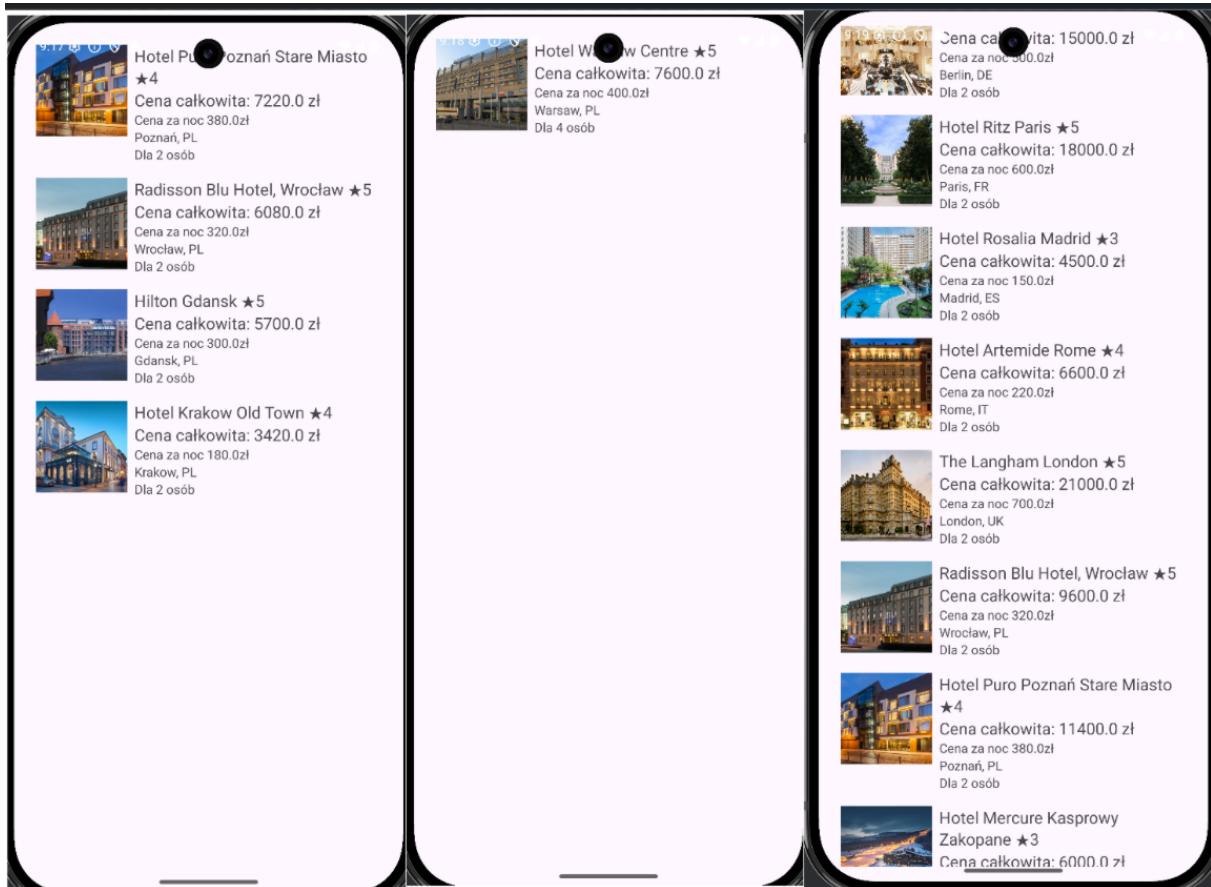
Rysunek 4. Widok wyszukiwania cz. 1



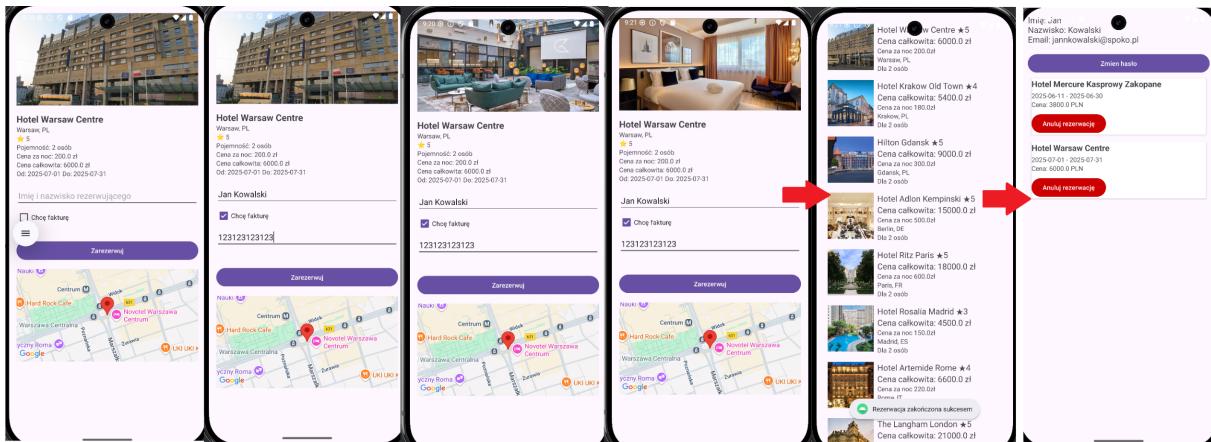
Rysunek 5. Widok wyszukiwania cz. 2



Rysunek 6. Widok moje konto

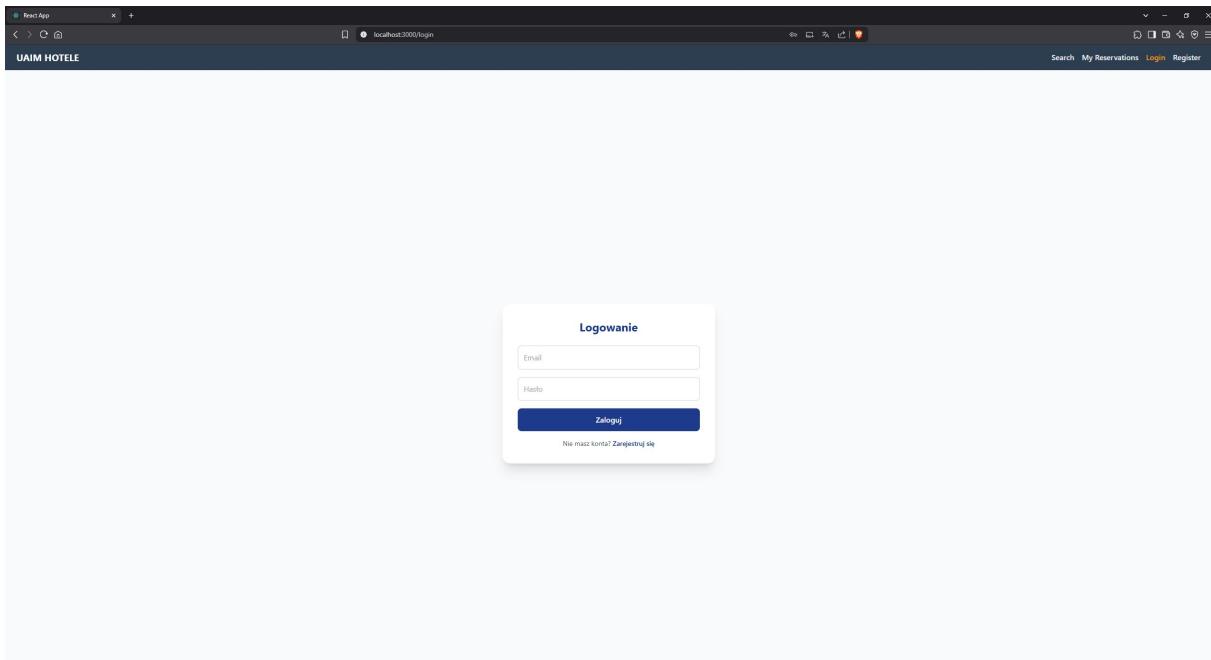


Rysunek 7. Wyniki wyszukiwania

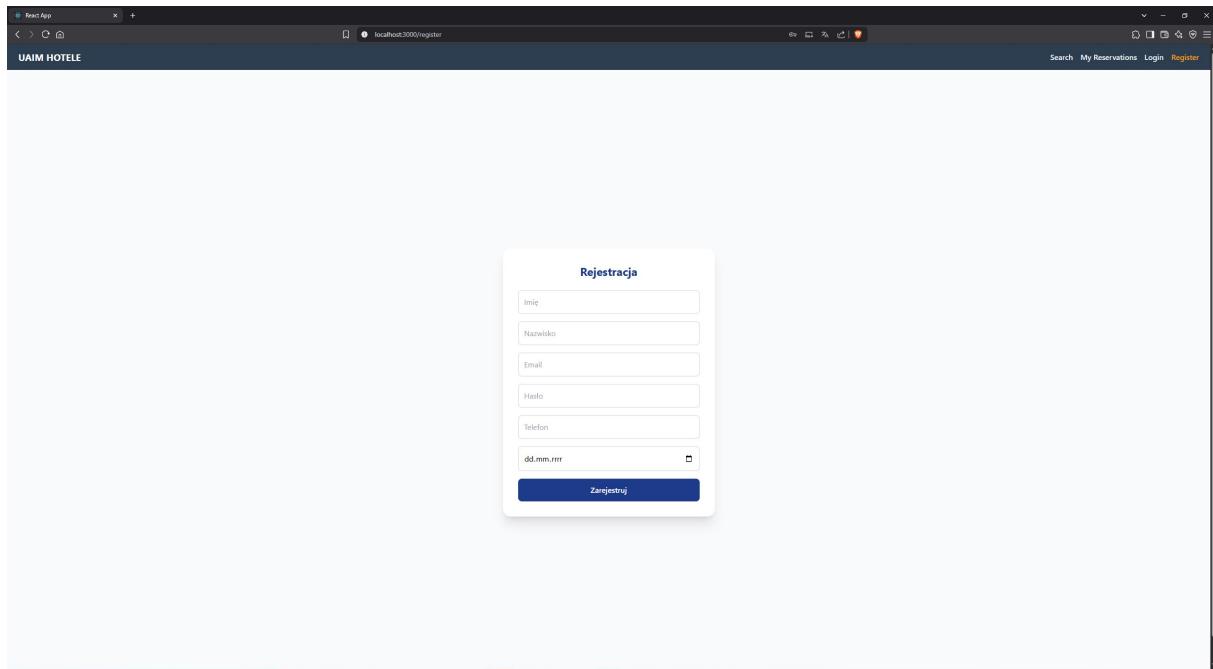


Rysunek 8. Widok pojedynczej oferty i rezerwowanie

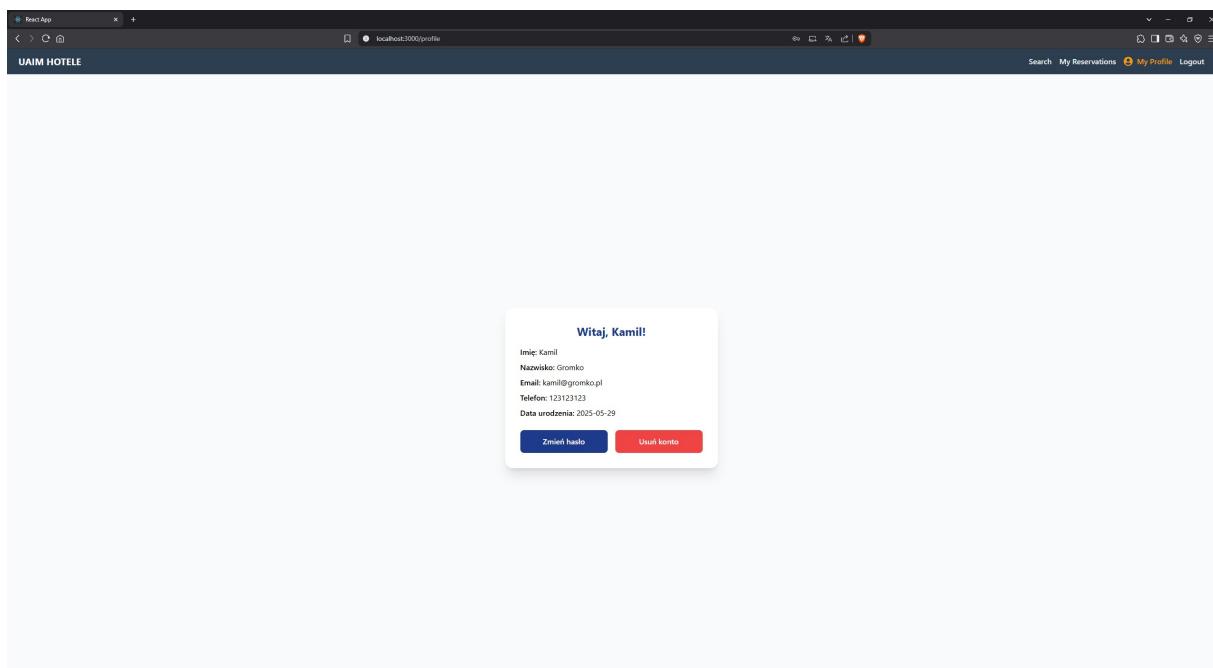
8.2. Aplikacja webowa



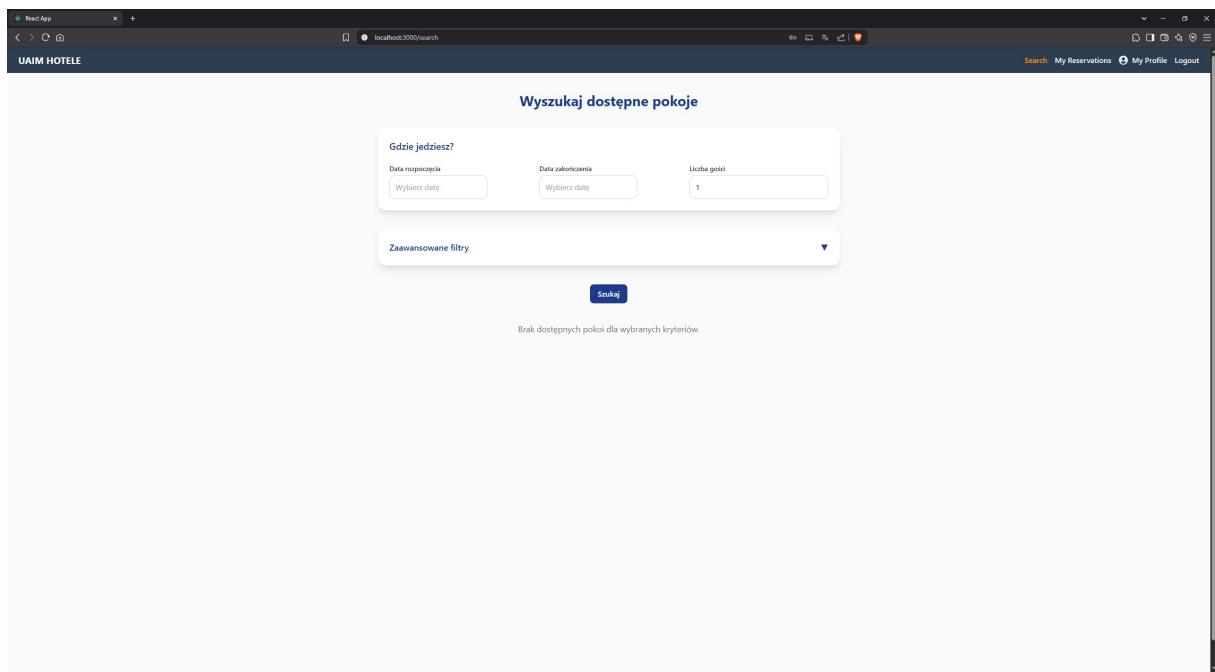
Rysunek 9. Logowanie



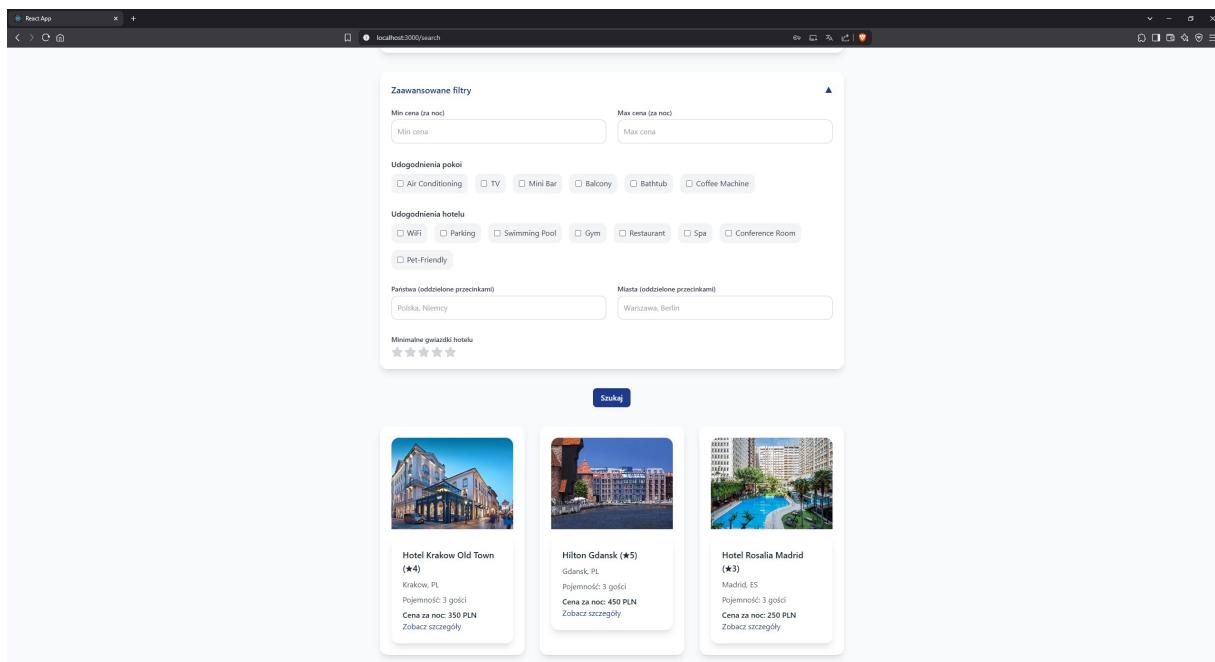
Rysunek 10. Rejestracja



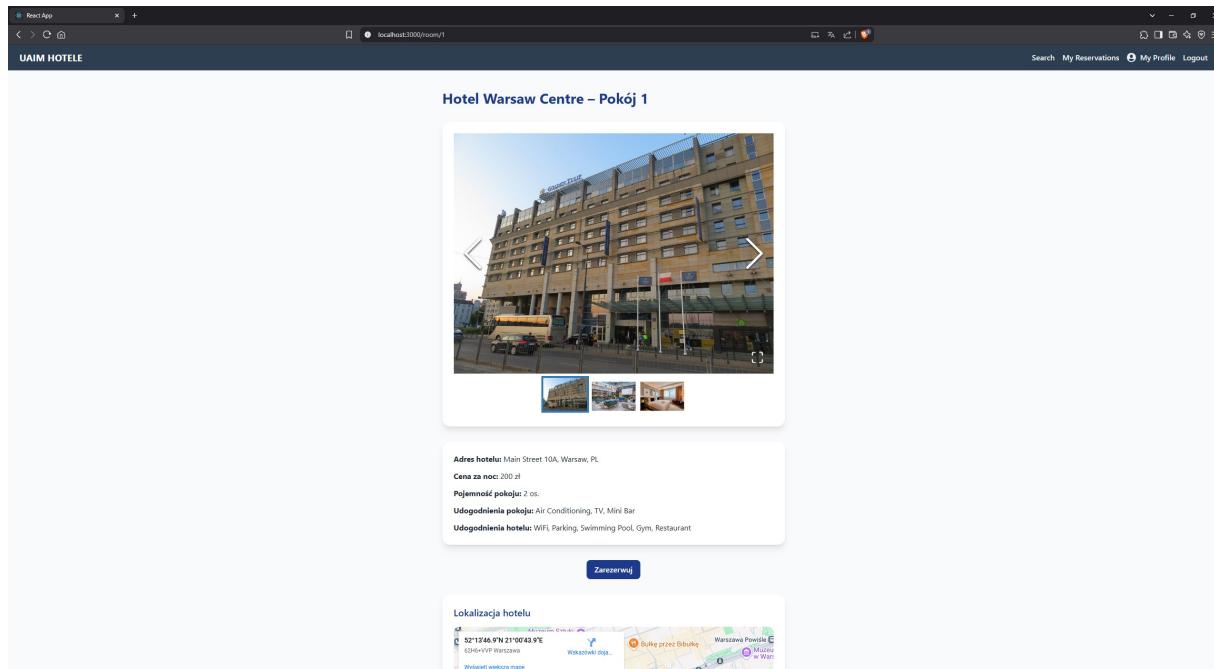
Rysunek 11. Podgląd profilu



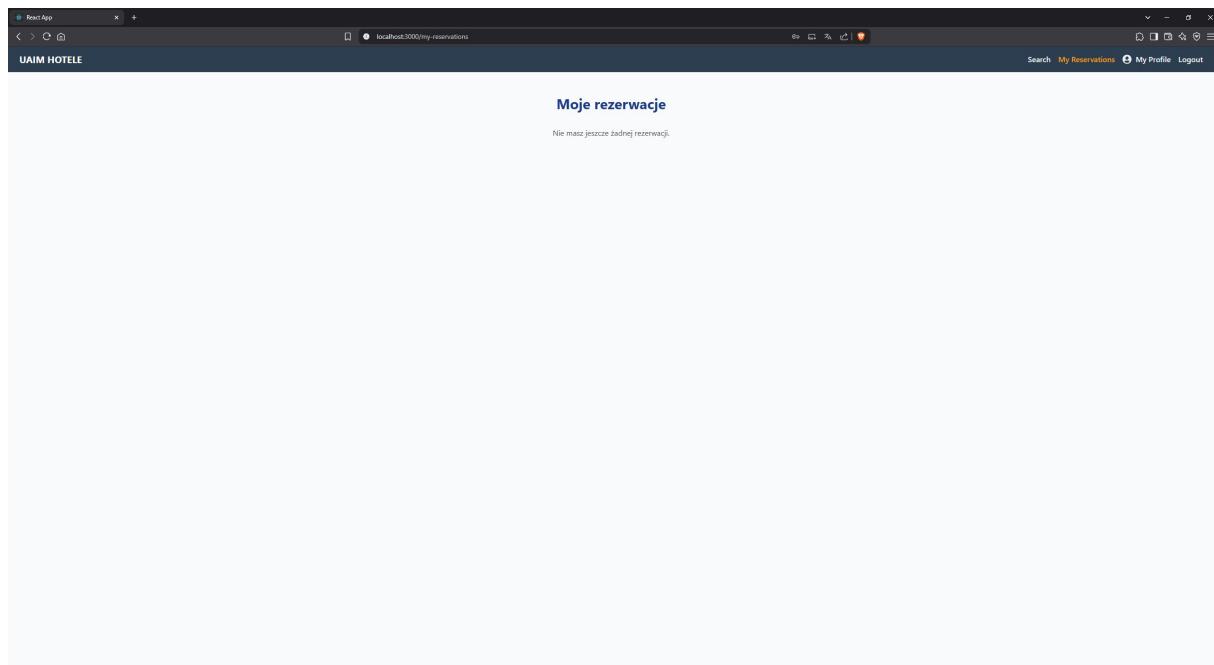
Rysunek 12. Wyszukiwarka



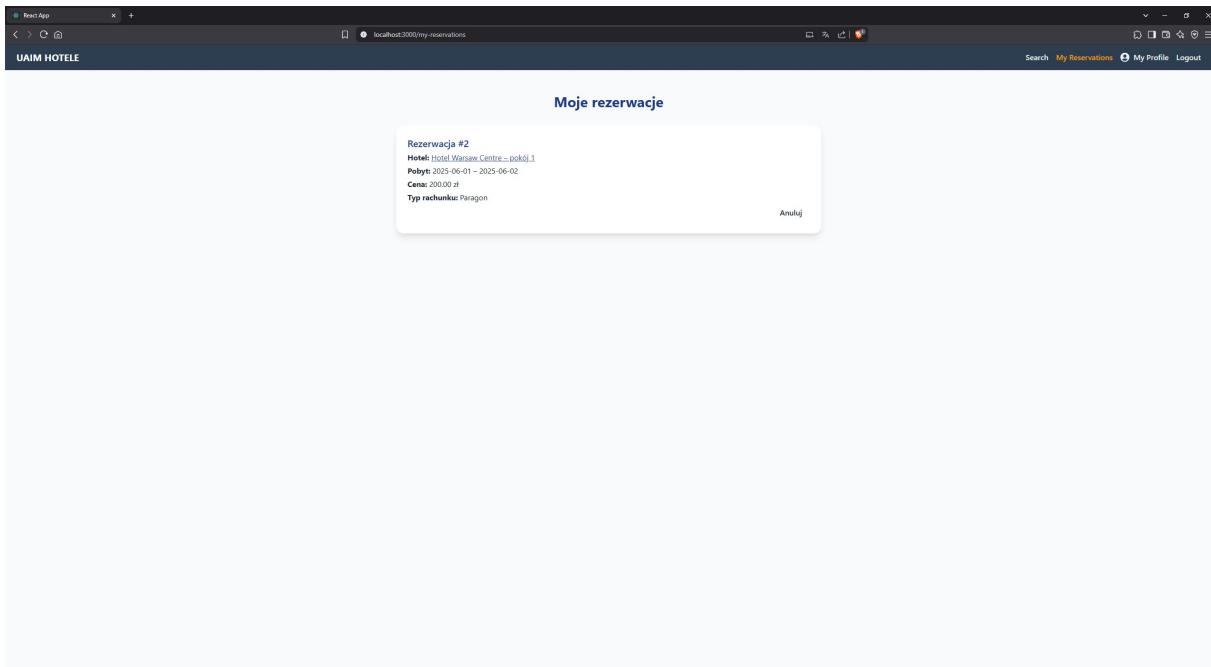
Rysunek 13. Wyniki i filtry



Rysunek 14. Widok wybranego hotelu z mapą



Rysunek 15. Widok pustych rezerwacji



Rysunek 16. Widok z rezerwacją

9. Wnioski

Zrealizowany projekt stanowi pełną implementację systemu do rezerwacji miejsc hotelowych, zgodnie z wymaganiami funkcjonalnymi przedstawionymi w specyfikacji przedmiotu. System został zaprojektowany modułowo i składa się z backendu napisanego w **Pythonie (Flask)**, frontendowej aplikacji webowej zrealizowanej w **React**, natywnej aplikacji mobilnej na **Androida** oraz bazy danych **PostgreSQL**. Wszystkie komponenty, z wyjątkiem aplikacji mobilnej, zostały zintegrowane za pomocą narzędzia **Docker Compose**.

Aplikacja frontendowa, zbudowana jako dynamiczna **SPA (Single Page Application)** w bibliotece **React**, oferuje nowoczesny i w pełni responsywny interfejs użytkownika. Zastosowano w niej architekturę komponentową, a do zarządzania nawigacją oraz stanem globalnym (autoryzacją) wykorzystano odpowiednio biblioteki **React Router DOM** oraz **React Context API**. Zaimplementowane funkcje obejmują m.in. rejestrację i logowanie użytkownika, wyszukiwanie ofert z wieloma filtrami (lokalizacja, daty, udogodnienia, liczba gwiazdek), tworzenie i anulowanie rezerwacji, a także automatyczne generowanie i wysyłanie e-maili z dokumentami potwierdzającymi (paragon lub faktura).

Funkcjonalność została zweryfikowana testami jednostkowymi. Testy objęły zarówno aplikację frontendową, gdzie osiągnięto pokrycie kodu na poziomie ~90%, jak i główne moduły backendu, gdzie pokrycie wyniosło 96%, co w obu przypadkach znacząco przekracza projektowe wymagania i potwierdza wysoką jakość implementacji.

Projekt cechuje się przejrzystą strukturą katalogów, czytelnym kodem oraz zgodnością ze standardami wymaganymi w środowisku akademickim. Opracowane rozwiązanie może być w przyszłości łatwo rozszerzane o nowe funkcje, takie jak system ocen, płatności online czy zarządzanie opiniami. Całość została przygotowana zgodnie z założeniami projektowymi i udokumentowana we wszystkich wymaganych obszarach.