



POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Imię Nazwisko

Mateusz Rosiński s28004

Grzegorz Szczęsny s27881

System do zarządzania najmem nieruchomości w architekturze mikroservisowej i podejściu domenowym (DDD)

Praca inżynierska napisana pod
kierunkiem:

Dr. Inż. Mariusza Trzaski, prof.
PJATK

Warszawa, lipiec 2026

Streszczenie

Praca dotyczy zaprojektowania i implementacji rozproszonego systemu do zarządzania procesem najmu nieruchomości krótko i długoterminowym. Jest oparta na nowoczesnej architekturze mikroserwisowej. Rozwiązania dostępne na dzisiejszym rynku często nie nadążają nad dynamicznym rozwojem najmów, zarówno krótkoterminowych, jak długoterminowych. Wymagająca branża, mnogość skomplikowanych procesów biznesowych, automatyzacja powtarzających się czynności wymagają opracowania nowego rozwiązania, które będzie wspierało zarządzanie codziennymi czynnościami i ułatwi biznes.

W ramach pracy zaprojektowano i zaimplementowano system, który składa się z wielu niezależnych mikroserwisów komunikujących się za pośrednictwem synchronicznego REST API oraz brokera wiadomości RabbitMQ. Warstwa serwisowa została stworzona przy użyciu Java Spring Boot oraz .NET Core, natomiast frontend został utworzony przy użyciu Angulara. Całość środowiska została zbudowana przy użyciu Dockera oraz orkiestrowana za pomocą dystrybucji Kubernetes K3s.

System zapewnia obsługę kluczowych procesów: zarządzanie lokalami, automatyczne generowanie umów najmu, weryfikację firm najemców przez integrację z rejestrami GUS, rozliczenia płatności, a także powiadomienia e-mail. Bezpieczeństwo realizowane jest poprzez własny serwis autoryzacyjny z tokenami JWT weryfikowanymi kluczami asymetrycznymi.

Rezultatem pracy jest działający prototyp na środowisku postawiony na wirtualnej maszynie w sieci uczelni, który potwierdza wykonalność przyjętej architektury i stanowiący podstawę do dalszego rozwoju jako komercyjne rozwiązanie dla firm i osób fizycznych zajmujących się zarządzaniem nieruchomościami na wynajem.

Słowa kluczowe: Najem krótkoterminowy, najem długoterminowy, Wynajmy, REST API, Angular, Java, C#, .NET, Spring Boot, Rabbit

Podziękowania

Autorzy pracy wyrażają podziękowania promotorowi za cenne wskazówki merytoryczne oraz konstruktywne uwagi, które miały istotny wpływ na kształt niniejszej pracy. Podziękowania kierowane są również do firm, w których autorzy zdobywali doświadczenie zawodowe związane z projektowaniem i implementacją systemów klasy Enterprise. Zdobyte kompetencje i wiedza praktyczna znacząco wpłynęły na poziom techniczny prezentowanych rozwiązań.

Spis treści

1. WSTĘP	9
1.1. Cel pracy	9
1.2. Rezultat pracy.....	10
1.3. Rozwiązania przyjęte w pracy.....	11
1.4. Zakres pracy	12
1.5. Organizacja pracy.....	13
1.6. Podział prac projektowych	14
2. ANALIZA ROZWIĄZAŃ DOSTĘPNYCH NA RYNKU	17
2.1. Aplikacja „Rentio”	17
2.2. Aplikacja „Apartemo”	18
2.3. Aplikacja „FlatFlow”	19
2.4. Aplikacja „RentFlow”	19
2.5. Aplikacja „Buildium”.....	20
2.6. Podsumowanie	20
3. METODOLOGIA WYTWARZANIA OPROGRAMOWANIA.....	22
3.1. Scrum	22
3.2. Jira	22
3.3. Code review (weryfikacja jakości kodu).....	23
4. PROPOZYCJA SYSTEMU DO ZARZĄDZANIA NAJMEM KRÓTKO ORAZ DŁUGOTERMINOWYM	24
4.1. Wizja systemu	24
4.2. Specyfikacja wymagań funkcjonalnych	25
4.2.1 Zarządzanie lokalami	25
4.2.2 Zarządzanie kontami i uprawnieniami	26
4.2.3 Obsługa umów najmu oraz ich procesowanie.....	27
4.2.4 System powiadomień	27
4.2.5 System pomiarów liczników w mieszkaniach	28
4.2.6 Rejestracja i logowanie.....	29
4.2.7 Cykliczne generowanie należności	29
4.2.8 Dostępność nieruchomości.....	29
4.2.9 Generowanie dokumentów PDF	30
4.3. Specyfikacja wymagań poza funkcjonalnych	30
4.3.1 Bezpieczeństwo i autoryzacja	31
4.3.2 Skalowalność systemu	33
4.3.3 Wydajność	35
4.3.4 Niezawodność.....	37
4.3.5 Integracje zewnętrzne.....	38
5. ZASTOSOWANE TECHNOLOGIE, NARZĘDZIA I STANDARDY.....	39
5.1. Użyte technologie.....	39
5.1.1 Java	39

5.1.2	<i>Spring Boot</i>	39
5.1.3	<i>C#</i>	40
5.1.4	<i>.NET</i>	40
5.1.5	<i>PostgreSQL</i>	41
5.1.6	<i>Angular</i>	41
5.1.7	<i>Typescript</i>	42
5.1.8	<i>RabbitMQ</i>	42
5.2.	Narzędzia i biblioteki	42
5.2.1	<i>Leaflet.js</i>	42
5.2.2	<i>RxJs</i>	43
5.2.3	<i>NgRx</i>	43
5.2.4	<i>Mockito</i>	43
5.2.5	<i>Liquibase</i>	44
5.2.6	<i>PuppeteerSharp</i>	45
5.2.7	<i>Entity Framework</i>	45
5.2.8	<i>Hibernate</i>	46
5.2.9	<i>Docker</i>	46
5.2.10	<i>Kubernetes</i>	48
5.2.11	<i>Traefik</i>	49
5.2.12	<i>Gitlab CI/CD</i>	49
5.2.13	<i>Docker DinD</i>	51
5.2.14	<i>Caffeine</i>	51
5.2.15	<i>Swagger</i>	52
5.2.16	<i>Lombok</i>	53
5.2.17	<i>Postman</i>	53
5.2.18	<i>Thymeleaf</i>	54
5.3.	Zastosowane standardy	54
5.3.1	<i>HTTP</i>	55
5.3.2	<i>SOAP</i>	55
5.3.3	<i>Architektura sterowana zdarzeniami (Event Driven Architecture)</i>	56
6.	PROJEKT I IMPLEMENTACJA	57
6.1.	Architektura mikroservisowa	57
6.1.1	<i>Porównanie pomiędzy mikroservisem i monolitem</i>	58
6.1.2	<i>Porównanie wzorców architektonicznych współczesnych systemów</i>	60
6.2.	Podział na poszczególne mikroservisy w projekcie	68
6.3.	Architektura REST API	70
6.3.1	<i>Komunikacja synchroniczna kontra asynchroniczna</i>	71
6.4.	Architektura tokenów uwierzytelniających.....	74
6.4.1	<i>Klucze asymetryczne do podpisywania</i>	76
6.4.2	<i>OAuth 2.0</i>	76
6.4.3	<i>OAuth 2.0 vs klasyczna autoryzacja</i>	77
6.4.4	<i>JWT</i>	77
6.4.5	<i>OWASP</i>	78
6.5.	Zastosowana architektura bazy danych w systemie	79
6.5.1	<i>Relacyjne bazy danych</i>	79
6.5.2	<i>Porównanie bazy danych współdzielonej, a oddzielnej dla każdej domeny lub mikroservisu</i>	79

6.5.3	<i>Uwzględnione optymalizacje bazy</i>	80
6.5.4	<i>Indeksy</i>	80
6.5.5	<i>Zwykłe (pojedyncze)</i>	81
6.5.6	<i>Złożone</i>	81
6.5.7	<i>Rozszerzenie indexów Btree GiSTs do bazy PostgreSQL</i>	81
6.6.	DDD (Domain-Driven-Design)	82
6.6.1	<i>Model domenowy</i>	83
6.6.2	<i>Bounded Context (Ograniczony kontekst)</i>	85
6.6.3	<i>Wspólny język biznesowy</i>	86
6.6.4	<i>Warstwy architektury DDD</i>	86
6.6.5	<i>Zalety rozwiązania</i>	86
6.6.6	<i>Wady rozwiązania</i>	87
6.7.	Zastosowane wzorce projektowe	88
6.7.1	<i>Builder</i>	88
6.7.2	<i>ACL</i>	89
6.7.3	<i>Adapter</i>	90
6.7.4	<i>Strategia</i>	91
6.7.5	<i>Fabryka</i>	92
6.8.	Metodologia SOLID	93
6.8.1	<i>Single Responsibility Principle (SRP)</i>	93
6.8.2	<i>Open/Closed Principle (OCP)</i>	93
6.8.3	<i>Liskov Substitution Principle (LSP)</i>	94
6.8.4	<i>Interface Segregation Principle (ISP)</i>	94
6.8.5	<i>Dependency Inversion Principle (DIP)</i>	95
6.9.	Implementacja aplikacji webowych - frontend	95
6.9.1	<i>Prywatne współdzielona biblioteka npm dla aplikacji frontendowych</i>	95
6.9.2	<i>The Angular Component Development Kit</i>	99
6.9.3	<i>The Angular Material</i>	100
6.10.	Implementacja aplikacji backendowych	102
6.10.1	<i>Współdzielone API dla serwisów</i>	102
6.10.2	<i>BFF - Backend for frontend</i>	104
6.10.3	<i>Authorization Service</i>	106
6.10.4	<i>Generowanie dokumentów</i>	109
6.10.5	<i>Mechanizm streamingu i double batch</i>	110
6.10.6	<i>Mechanizm mierników zasobów</i>	112
6.10.7	<i>Obsługa asynchroniczna wielu wątków</i>	113
6.10.8	<i>Optymalizacja requestów poprzez batchowe odpytanie danych</i>	114
6.10.9	<i>Mechanizm obsługi powiadomień</i>	114
6.11.	DTO	115
6.12.	JSON	116
6.13.	Część wdrożeniowa systemu	116
6.13.1	<i>Lokalne środowisko</i>	116
6.13.2	<i>Środowisko produkcyjne</i>	117
6.13.3	<i>Przekierowanie ruchu na poszczególne domeny</i>	118
6.13.4	<i>Podsumowanie</i>	118
7.	INTERFEJS UŻYTKOWNIKA	119
7.1.	Aplikacja webowa zarządcy	119

7.1.1	<i>Ekran logowania</i>	119
7.1.2	<i>Ekran rejestracji i logowania zarządcy głównego</i>	120
7.1.3	<i>Ekran główny</i>	124
7.1.4	<i>Ekran ustawień</i>	125
7.1.5	<i>Ekran kreatora najmu</i>	126
7.1.6	<i>Ekran administratora</i>	128
7.1.7	<i>Ekran listy najmów</i>	128
7.1.8	<i>Ekran szczegółów najmu</i>	129
7.1.9	<i>Ekran opłat</i>	130
7.1.10	<i>Ekran liczników/mierników</i>	130
7.1.11	<i>Ekran dodawania/modyfikacji nieruchomości</i>	131
7.1.12	<i>Ekran historii najmów dla nieruchomości</i>	132
7.2.	<i>Aplikacja webowa najemcy</i>	133
7.2.1	<i>Ekran logowania</i>	133
7.2.2	<i>Ekran listy najmów</i>	134
7.2.3	<i>Ekran listy należności</i>	134
7.2.4	<i>Ekran szczegółów najmu</i>	135
7.2.5	<i>Ekran płatności</i>	136
7.3.	<i>Widoki współdzielone pomiędzy aplikacjami frontendowymi</i>	136
7.3.1	<i>Ekran ustawienia hasła użytkownika</i>	137
7.3.2	<i>Ekran weryfikacji użytkownika po kodzie OTP</i>	137
7.3.3	<i>Ekran weryfikacji użytkownika po tokenie</i>	139
8.	TESTY	142
8.1.	<i>Testy jednostkowe</i>	142
8.1.1	<i>Konfiguracja</i>	142
8.1.2	<i>Spy - częściowe mockowanie</i>	143
9.	PODSUMOWANIE	144
9.1.	<i>Wady i zalety rozwiązania</i>	144
9.2.	<i>Potencjalne kierunki rozwoju</i>	145
9.3.	<i>Wnioski</i>	146
PRACE CYTOWANE		148
SPIS LISTINGÓW		153
SPIS RYSUNKÓW		155

1. Wstęp

Najmy w ostatnich latach stały się bardzo popularną i łatwo dostępną formą korzystania z mieszkań. Coraz więcej firm oraz osób prywatnych decyduje się na zakup mieszkania bądź kilku mieszkań, odnowienie ich i udostępnienie w celu wynajęcia, żeby generować tym samym zyski. Takie rozwiązanie przynosi korzyści obu stronom. Zarządcy czerpią z niego przede wszystkim wartość finansową, natomiast najemcy zyskują możliwość zamieszkania bliżej centrum miasta bez konieczności zakupu nieruchomości, co przekłada się również na oszczędności związane z dojazdem do pracy.

Wzrost najmów w dużej mierze dziś bierze się ze zwiększenia liczby mieszkań, liczby osiedli których z roku na rok pojawia się coraz więcej. Obserwowany od kilku lat wzrostowy trend na rynku nieruchomości utrzymuje się, pomimo medialnych doniesień o rosnących cenach mieszkań. Pomimo utrzymujących się wysokich cen mieszkań, popyt na lokale mieszkalne pozostaje relatywnie wysoki. Potwierdzają to dane Narodowego Banku Polskiego wskazujące na wzrost liczby transakcji mieszkaniowych [1] oraz dalszy wzrost cen w ujęciu rocznym. Dodatkowo wzrost wynagrodzeń gospodarstw domowych oraz zwiększona aktywność inwestycyjna deweloperów świadczą o utrzymującej się atrakcyjności rynku mieszkaniowego [2].

Żeby wspomóc w działaniu każdą ze stron tego rynku najmów, dużą pomocą byłaby automatyzacja konkretnych procesów biznesowych w tej dziedzinie. Automatyzacja pozwoliłaby na realizację poszczególnych etapów umowy najmu oraz przejęcie odpowiedzialności za rutynowe czynności administracyjne. Przykładem takiej czynności jest obsługa procedur formalnych po stronie firmy i klienta, która usunie konieczność angażowania zasobów ludzkich. Zwolnienie zarządców od tych obowiązków stworzyłoby przestrzeń do skupienia się na działaniach o wyższej wartości biznesowej. Takimi działaniami są: rozwój oferty, budowanie marki, pozyskiwanie nowych kontraktów oraz inwestycje bezpośrednio przekładające się na wzrost liczby klientów i zysków.

1.1. Cel pracy

Celem prezentowanej pracy jest zaprojektowanie i implementacja dedykowanego systemu informatycznego wspierającego zarządzanie procesem najmu nieruchomości. System obejmuje kompleksową obsługę zarówno najmów krótkoterminowych, jak i długoterminowych, centralizując kluczowe procesy biznesowe firm oraz osób prywatnych działających w tej branży. System umożliwi sprawne przeprowadzenie zarówno zarządców nieruchomości, jak i ich klientów (najemców) przez cały cykl życia umowy najmu - od momentu jej zawarcia, poprzez bieżącą obsługę i monitorowanie, aż do jej zakończenia. Projektowane rozwiązanie stawia na centralizację wszystkich kluczowych procesów związanych z zarządzaniem najmem w jednym, spójnym środowisku informatycznym. Dzięki temu zarówno przedsiębiorstwa, jak i użytkownicy indywidualni uzyskają dostęp jednego narzędzia, które pozwoli na efektywne zarządzanie dokumentacją, komunikacją, płatnościami oraz prowadzeniem statusów poszczególnych umów.

Zastosowanie systemu przyczyni się do zwiększenia efektywności operacyjnej poprzez automatyzację powtarzalnych czynności, poprawę przejrzystości procesów oraz ograniczenie ryzyka błędów wynikających z rozproszenia danych. Zarządcy i agenci nieruchomości zyskają możliwość szybszego reagowania na pojawiające się problemy, bieżącego monitorowania statusu najmu oraz generowania raportów i analiz wspierających podejmowanie decyzji.

Z kolei najemcy otrzymają dostęp do intuicyjnego narzędzia umożliwiającego zarządzanie swoim najmem w jednym miejscu, w tym wgląd do dokumentów, historii płatności czy zgłoszeń serwisowych. System zapewnia również usprawnioną komunikację z zarządcą, co przełoży się na wyższy poziom obsługi oraz większą przejrzystość relacji między stronami. Projektowany system kładzie szczególny nacisk na wysoką konfigurowalność oraz możliwość dostosowania do różnych modeli biznesowych związanych z najmem nieruchomości oraz lokali. Istotnym założeniem jest również zapewnienie kompatybilności rozwiązania, umożliwiającej integrację z zewnętrznymi systemami oraz usługami. Dzięki zastosowaniu odpowiednich interfejsów komunikacyjnych możliwe jest rozszerzenie funkcjonalności systemu bez konieczności jego przebudowy.

System został zaprojektowany z uwzględnieniem wydajności oraz optymalizacji operacji na danych, co pozwala na jego sprawne działanie przy dużej liczbie obsługiwanych umów i użytkowników. Zastosowane podejście umożliwia efektywne zarządzanie zasobami oraz minimalizację opóźnień w dostępie do informacji.

1.2. Rezultat pracy

Jako końcowe rozwiązanie założonego celu jest opracowanie i wdrożenie webowego funkcjonalnego systemu CRM¹, który centralizuje dane oraz automatyzuje wewnętrzne procesy firm i osób prywatnych zajmujących się najem, jak również przyspiesza realizację najmu oraz tym samym zwiększa zadowolenie klientów poprzez dawanie im większej wartości w postaci bliższego, szybszego i dokładniejszego opiekowania się w trakcie procesu.

Wspomniany system umożliwia odciążenie zarządców od spraw prostszych oraz powtarzalnych. Dzięki temu więcej czasu może zostać poświęcone na zwiększenie relacji z klientem co pozwoli na wzrost nowych najemców oraz skupienie się na rzeczach kluczowych dla firmy. Dodatkowym, skutkiem tego rozwiązania będzie większa ilość czasu na podejmowanie wymagających szczególnych decyzji procesowych. System koncentruje w jednym miejscu wszystkie kluczowe funkcjonalności niezbędne do obsługi najmu. Użytkownik ma natychmiastowy dostęp do pełnej dokumentacji elektronicznej powiązanej z konkretną umową, bez konieczności przeszukiwania rozproszonych źródeł.

Najważniejszą funkcją systemu jest możliwość prowadzenia najemcy przez kompletny cykl najmu. Trwa on od momentu zawarcia umowy aż do jej zakończenia. Zapewnia to ciągłość obsługi płatności, obiegu dokumentów oraz bieżącego monitorowania statusu najmu.

¹ Customer Relationship Management - z. ang. Scentralizowane oprogramowanie do zarządzania relacjami z klientami

1.3. Rozwiązania przyjęte w pracy

W pracy zdecydowano się na zastosowanie architektury rozproszonej opartej na podejściu domenowym DDD², w ramach której zarówno poszczególne mikroserwisy jak i ich struktura wewnętrzna zostały zaprojektowane w oparciu o wyodrębnione domeny biznesowe. Każda z domen koncentruje się na realizacji określonej odpowiedzialności biznesowej, co pozwala na zwiększenie skalowalności systemu oraz lepsze odwzorowanie procesów biznesowych. System został podzielony na dwie główne domeny biznesowe:

- najemcy (z. ang *tenant*),
- zarządcy i agenci nieruchomości (z. ang *landlord*).

Każda z tych domen została następnie rozłożona na mniejsze poddomeny biznesowe odpowiedzialne za konkretne obszary funkcjonalne systemu, takie jak zarządzanie mieszkaniami, obsługa płatności oraz między innymi najmy. Takie podejście umożliwia wyraźne rozdzielenie logiki biznesowej oraz ograniczenie zależności pomiędzy poszczególnymi komponentami systemu. Zastosowanie takiego podejścia z architekturą rozproszoną pozwala na wydzielenie niezależnych kontekstów biznesowych³, co przekłada się na możliwość niezależnego rozwijania, testowania oraz wdrażania poszczególnych modułów. Dodatkowo wpływa to pozytywnie na skalowalność rozwiązania oraz większą odporność na błędy i zmiany w obrębie pojedynczych komponentów.

- Aplikacje klienckie - dwie niezależnie responsywne⁴ aplikacje przeglądarkowe wystawione na oddzielnych portach napisane w technologii Angular w wersji 19.2 oraz TypeScript w wersji 5.7.2,
- Prywatna biblioteka frontendowa - biblioteka npm⁵, która posiada współdzielone komponenty, typy wyliczeniowe i schematy pomiędzy aplikacjami klienckimi, zaimplementowana w Angular 19.2 oraz Angular CDK,
- Warstwa serwerowa - 8 niezależnych domenowych serwisów biznesowych które wystawiają WebAPI⁶ w technologii REST korzystając z protokołu HTTP, SOAP oraz zdarzeń do komunikacji oraz 4 modele API, posiadające obiekty i struktury współdzielone,
 - 6 serwisów zaimplementowanych w Java 21 na frameworku⁷ Spring 6.2.11 i SpringBoot w wersji 3.5.6,
 - 2 serwisy zaimplementowane w C# w frameworku ASP .NET Core 9.0,
 - 4 modele API zaimplementowane w Spring,
- Baza danych Postgres użyta w projekcie w wersji 17,
- Mechanizm zdarzeń zaimplementowany w technologii RabbitMQ 4.2 wykorzystywany do operacji z większą ilością danych które muszą zostać przesłane między serwisami,

² Domain Driven Development - z. ang podejście w projektowaniu oprogramowania. Które koncentruje się na modelowaniu systemu wokół domeny biznesowej i jej logiki

³ Bounded Context (Kontekst Ograniczony) - wyraźnie wydzielony obszar systemu, w którym obowiązuje spójny model domenowy oraz jednolity język biznesowy

⁴ Cecha interfejsu polegająca na dostosowaniu jego wyglądu i układu do różnych rozmiarów ekranów oraz urządzeń

⁵ Node Package Manager - domyślny manager pakietów środowiska Node.js, służący do instalowania

⁶ Interfejs komunikacyjny

⁷ Zestaw narzędzi bibliotek i konwencji programistycznych ułatwiających tworzenie aplikacji

- Serwer obiektowy do przechowywania plików, dokumentów i zdjęć MinIO.

Obrazy aplikacji są przechowywane na koncie Gitlab w zasobach registry⁸ i są budowane w trakcie wdrażania kodu na gałęzi master⁹ wykorzystując procesy CI/CD¹⁰ platformy Gitlab.

System został wdrożony na serwer przy pomocy konteneryzacji i orkiestracji z wykorzystaniem technologii Kubernetes oraz Docker, co umożliwia łatwe wdrożenie systemu na każde środowisko, na dowolnej maszynie zarówno Linux, Windows czy MacOS.

1.4. Zakres pracy

W ramach realizacji pracy zostały przeanalizowane podstawowe procesy najmu nieruchomości pozwalające na przeprowadzenie najemcy przez cały cykl najmu - od zawarcia umowy do jej zakończenia. Na początkowym etapie szczególną uwagę poświęcono wymaganiom użytkownika, które muszą zostać spełnione, żeby system mógł sprawnie funkcjonować i móc realnie wspomóc pracę współczesnych zarządców nieruchomości oraz usprawnić obsługę umów po stronie najemców.

W tym celu przeprowadzono analizę istniejących rozwiązań dostępnych na rynku, co pozwoliło lepiej zrozumieć zachowania, potrzeby oraz oczekiwania użytkowników. Wyniki tych badań umożliwiły rozpoczęcie spotkań, podczas których analizowano możliwe przypadki użycia oraz rozwiązania implementacyjne jakie możemy zastosować dla każdego procesu oraz modułów systemu. Po wstępnych ustaleniach została opracowana architektura systemu, oparta na modelu rozproszonym z wykorzystaniem mikroservisów, podzielonych według głównych domen biznesowych. Takie podejście ułatwia dalszy rozwój systemu, zwiększa jego modularność oraz pozwala na łatwiejsze dołączanie nowych modułów i rozszerzanie istniejących funkcjonalności.

Prace implementacyjne zostały rozpoczęte od stworzenia serwisu autoryzacyjnego odpowiedzialnego za wydawanie tokenów JWT¹¹, uwierzytelnianie oraz weryfikację użytkowników. Cały mechanizm autoryzacji oraz autentykacji opiera się na rolach użytkowników oraz wykorzystaniu klucza asymetrycznego. Warstwa użytkownika składa się z dwóch aplikacji klienckich, dostępnych pod oddzielnymi adresami w zależności od domeny biznesowej. Obie aplikacje korzystają ze wspólnych schematów, komponentów oraz stylów umieszczonych w dedykowanej, prywatnej bibliotece frontendowej.

Komunikacja pomiędzy aplikacjami użytkownika, a warstwą serwerową odbywa się za pośrednictwem serwisów BFF¹², połączonych oddzielnie do każdej z aplikacji klienckich, przez które przechodzą wszystkie odpytania do serwera. Uwierzytelnienie w nich działa na podstawie

⁸ Rejestr - z. ang centralny system przechowywania i udostępniania zasobów, umożliwiający ich wersjonowanie i pobieranie

⁹ Główna linia w systemach kontroli wersji (Git), która zazwyczaj zawiera stabilna, produkcyjna wersja kodu systemu

¹⁰ Continuous Integration/Continuous Deployment - z. ang praktyka wytwarzania oprogramowania polegająca na automatycznym integrowaniu zmian w kodzie, testowaniu i wdrażaniu na środowiska produkcyjne

¹¹ JSON Web Token - z. ang Otwarty standard do bezpiecznego przekazywania informacji, pomiędzy stronami w postaci podpisanego, samodzielnego tokenu

¹² Backend For Frontend - z. ang Wzorzec architektoniczny oddzielnego serwisu backendowego dopasowanego pod konkretną frontendową aplikację

autoryzacji tokenami JWT, które analizuje dany serwis BFF i działa ono w oparciu o technologię OAuth2¹³.

Warstwa serwerowa, oprócz serwisów BFF, składa się jeszcze z kilku niezależnych serwisów domenowych współdzielących jedną bazę danych, gdzie każdy domenowy serwis korzysta tylko z tych encji, które odpowiadają konkretnym zadaniom biznesowym takiego serwisu. Serwisy po stronie backendu wykorzystują również współdzielone projekty API, zawierające obiekty transferu danych (DTO¹⁴), typy wyliczeniowe oraz inne wspólne elementy wykorzystywane w wielu miejscach systemu.

Komunikacja pomiędzy serwisami została zrealizowana w sposób hybrydowy. Większość integracji opiera się na architekturze REST¹⁵, natomiast w obszarach niewymagających ścisłej kontroli przepływu danych lub tam, gdzie istotna jest optymalizacja wydajności, zastosowano komunikację zdarzeniową z wykorzystaniem kolejek wiadomości.

System podczas procesowania najmu również generuje dokumenty na podstawie przesłanych danych, a przechowywane są one na własnym, hostowanym serwerze obiekowym¹⁶. W ramach realizacji projektu przygotowano również dedykowany serwis dokumentów odpowiedzialny za zarządzanie plikami, ich zapisywanie, pobieranie oraz kontrolę nad przesyłanymi zdjęciami. Rozwiązanie to umożliwia centralizację obsługi dokumentacji oraz łatwiejsze rozwijanie funkcjonalności związanych z obiegiem dokumentów w przyszłości. Na serwerze są przechowywane między innymi umowy, załączniki, zdjęcia nieruchomości oraz inne pliki wykorzystywane przez system.

Jako usprawnienie bezpieczeństwa użytkowników od strony prawnej, została zrealizowana integracja z serwerem Głównego Urzędu Statystycznego, która umożliwia pobieranie danych rejestrowych firmy oraz ich automatyczną weryfikację podczas zakładania konta firmowego w systemie. Na podstawie uzyskanych informacji system weryfikuje, czy dany podmiot może korzystać z platformy i realizować procesowanie umów najmu.

Istotnym elementem zakresu pracy było również przygotowanie testów jednostkowych. Zostały one zaimplementowane przede wszystkim w bardziej wymagających i złożonych obszarach systemu, gdzie logika biznesowa wymagała dodatkowej weryfikacji poprawności działania. Testy te pozwalają ograniczyć ryzyko błędów, ułatwiają dalszy rozwój aplikacji oraz zwiększają stabilność całego rozwiązania.

1.5. Organizacja pracy

Praca została podzielona pomiędzy dwie osoby, z której każdy otrzymywał konkretne zadania jakie realizował w ciągu dwutygodniowego sprintu. Zadania obejmowały w głównej mierze implementacje w kodzie, natomiast oprócz niej został poświęcony czas na analizę, budowę architektury, testy oraz spotkania, na których były omawiane kwestie architektoniczne oraz różnego rodzaju problemy. Spotkania odbywały się w co dwutygodniowych sprintach, natomiast pomiędzy tymi dwoma tygodniami były również spotkania, gdzie poruszane zostały

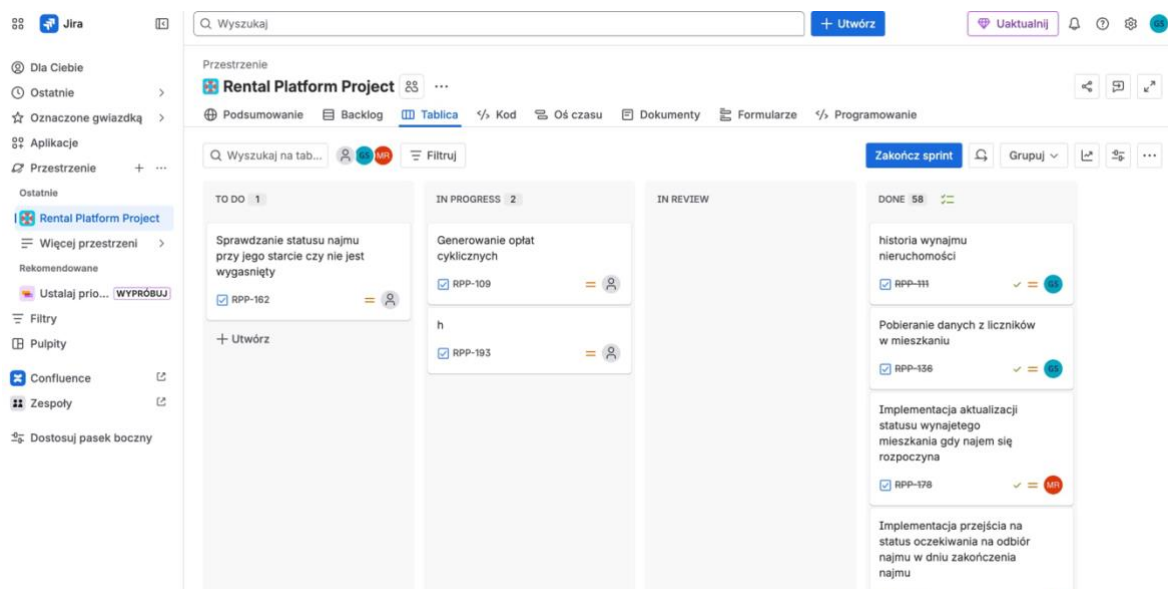
¹³ OAuth 2.0 - standard autoryzacji, który pozwala aplikacji uzyskać dostęp do zasobów bez podawania hasła

¹⁴ Data Transfer Object - z. ang obiekt służący do przenoszenia danych między warstwami aplikacji

¹⁵ Representational State Transfer - z. ang styl projektowania komunikacji, w którym odbywa się ona poprzez protokół http i operacje na zasobach

¹⁶ Serwer obiekowy (Object Storage Server) - z. ang system przechowywania danych, w którym informacje zapisane są w postaci obiektów zamiast w strukturze katalogów

problemy występujące w danym momencie u którejś z osób. Praca oraz planowanie zadań było dokumentowane na platformie Jira. Oprócz tego co jakiś czas były robione testy manualne sprawdzające poprawność logiki po wdrożeniu większych zmian implementacyjnych. Całość była koordynowana przez lidera grupy, który starał się, żeby wszystkie podjęte zadania zostały poprawnie zrobione i ukończone w ustalonym terminie.



Rysunek 1: Główne okno projektu na platformie Jira, źródło: opracowanie własne

1.6. Podział prac projektowych

W celu sprawnej realizacji projektu konieczne było odpowiednie rozdzielenie obowiązków pomiędzy członków zespołu. Podział prac projektowych został opracowany z uwzględnieniem kompetencji oraz zakresu odpowiedzialności poszczególnych osób, co pozwoliło na efektywną organizację działań i terminowe wykonanie wszystkich zadań. W niniejszym podrozdziale przedstawiono na tabeli 1 zakres prac realizowanych przez każdego członka zespołu.

Tabela 1: Tabela przedstawiająca podział prac w projekcie

Imię i nazwisko	Zrobione implementacje oraz rozdziały
Mateusz Rosiński	• Utworzenie serwisów oraz konfiguracja pod architekturę rozproszoną • Konfiguracja bazy danych • Konfiguracja kolejek Rabbitmq

	• Utworzenie flow provisioningu dla najemców i nowo utworzonych agentów/użytkowników przypisanych do panelu zarządcy • Utworzenie biblioteki frontendowej do współdzielenia komponentów frontendowych między projektami frontendowymi • Utworzenie współdzielonych konfiguracji oraz modeli (Data Transfer Object) w API REST • Konfiguracja liquibase do zarządzania bazą danych • Tworzenie ekranów frontendów • Moduł logowania, rejestracji i autoryzacji użytkowników (zarządców, najemców) • Bezpieczeństwo systemu IAM oraz role i uprawnienia, • Implementacja tokenów tymczasowych oraz systemowych i ich obsługa • Moduł tworzenia zarządcy oraz pobranie jego danych • Integracja z systemem GUS na szynie danych wykorzystującej SOAP • Moduł weryfikacji firmy, gdy użytkownik (zarządca bądź najemca) zakłada konto firmowe • Moduł najmu (inicjalizacja i przepływy flow po inicjalizacji) • Implementacja możliwości dodania konta bankowe dla zarządcy w celu wpłaty czynszów przez najemców • Integracja z serwisem generowania dokumentów • Integracja z serwisem do generowania płatności, należności • Utworzenie modułu użytkowników przypisanych pod konto zarządcy oraz zarządzaniem ich uprawnieniami dostępu oraz do zasobów • Panel zarządcy i jego podgląd danych
--	---

	• Konfiguracja docker-compose do budowania dockerowych kontenerów • Konfiguracja runnerów gitlabowych do pipeline'ów CI/CD • Konfiguracja podów oraz środowiska Kubernetes • Testy jednostkowe • Zarządzanie zespołem projektowym • Koordynowanie zadań projektowych • Rozdziały I, II, III, VI, VII, VIII
Grzegorz Szczęsny	• Utworzenie serwisu document-service (czyli nasz serwis do zarządzania dokumentami oraz zdjęciami) • Implementacja generowania, aktualizacji i pobierania dokumentów w document-service • Implementacja przesyłania i zapisu zdjęć wgranych na frontendzie • Konfiguracja modułu do przechowywania dokumentów MINIO • Moduł czytania danych z mierników/liczników w mieszkaniach oraz ich wyświetlanie w panelu zarządcy pod konkretnymi mieszkaniami • Moduł historii najmów dla konkretnej nieruchomości • Utworzenie serwisu do generowania płatności • Implementacja flow płatności w serwisie, który obsługuje płatność (payment-service) • Moduł rejestru lokali • Konfiguracja docker-compose do budowania dockerowych kontenerów • Konfiguracja runnerów gitlabowych do pipeline'ów CI/CD • Wyświetlanie najmów na frontendzie • Implementacja sidebar na frontendzie • Rozdziały IV, V, VI, VII, IX

2. Analiza rozwiązań dostępnych na rynku

Przed rozpoczęciem prac związanych z przygotowaniem i implementacją, warto zapoznać się z obecnym rynkiem w danej dziedzinie, żeby móc sprawnie opracować, wyliczyć zakres prac, koszty oraz terminy wdrożeń konkretnych modułów aplikacji. Najistotniejszą korzyścią płynącą z analizy rynku jest dogłębne poznanie jego specyfiki i aktualnych trendów. Wiedza ta stanowi bezpośrednią podstawę do zdefiniowania funkcjonalności systemu. Szczególną uwagę poświęcono tym funkcjom, których brakuje w istniejących rozwiązaniach lub które są dostępne wyłącznie w płatnych planach subskrypcyjnych. Takie funkcje bardzo często przyspieszają pracę i nadają jej jakość.

Przeprowadzone badanie rynku pozwoliło zidentyfikować zarówno zalety, jak i ograniczenia dostępnych aplikacji do zarządzania najmem. Warto zaznaczyć, że opisywany system ma charakter aplikacji wewnętrznej pomiędzy obecnymi systemami PSM¹⁷, a systemami do najmów krótkoterminowych, czyli tak zwanymi marketplace¹⁸. Oznacza to, że nie wchodzi w zakres platform z dziedziny Booking.com [3], Airbnb [4] czy Landfolk [5], dlatego nie będziemy ich tutaj przytaczać.

Jednakże pomimo tego, że nasza aplikacja nie jest typem marketplace, to posiada dużą konkurencję na rynku, ponieważ istnieją już rozwiązania typu PSM. Chcąc się wyróżnić na tle innych, nasza platforma jest czymś pomiędzy. Z jednej strony oferuje zarządcy wysoki poziom automatyzacji procesów, z drugiej zapewnia najemcy wygodny dostęp do opłat, dokumentów i bieżącego statusu najmu w jednym miejscu. Poniżej przedstawiono charakterystykę wybranych aplikacji dostępnych na rynku zarządzania najmem nieruchomości wraz z analizą ich zalet i ograniczeń w odniesieniu do opisywanego systemu.

2.1. Aplikacja „Rentio”

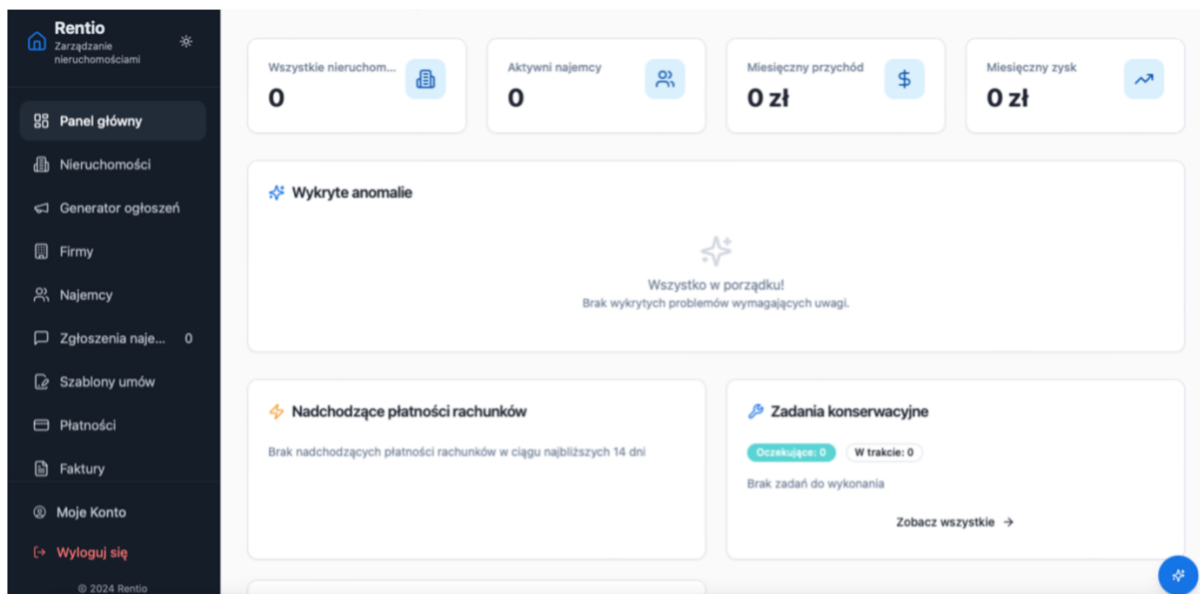
System [6] należący do kategorii PSM, który służy do kompleksowego zarządzania najmem mieszkaniowym. Dostępny jest na każdej platformie, gdzie zarządcy otrzymują system głównie na komputery osobiste, a najemcy korzystają z przeglądarki. Obejmuje on obsługę umów, ewidencje najemców, rozliczenia finansowe, fakturowanie oraz komunikację pomiędzy stronami. Aplikacja udostępnia szerokie możliwości operacyjne oraz jest bardzo dobrze dopasowana do warunków na polskim rynku, m.in. w zakresie czynszu, mediów i rozliczeń. Posiada integracje z AI¹⁹ oraz raportowanie faktur w systemie KSeF²⁰, które w dzisiejszej dobie są bardzo popularne i potrzebne przedsiębiorcy. Platforma posiada też kilka wad, głównie w postaci „ciężkiego systemu”, spowodowanego poprzez bardzo dużą ilość funkcjonalności, jak i również skomplikowane wdrożenie do aplikacji zarządcy i najemcy.

¹⁷ Property Management System - z. ang system informatyczny służący do zarządzania nieruchomościami

¹⁸ Internetowa lub fizyczna platforma handlowa, na której wielu sprzedawców oferuje swoje produkty lub usługi

¹⁹ Artificial Intelligence - z. ang Sztuczna inteligencja

²⁰ Krajowy System e-Faktur - system do przechowywania, odbierania i wystawiania sprzedażowych dokumentów elektronicznych



Rysunek 2: Zrzut ekranu pokazujący główny panel aplikacji Rentio, źródło: opracowanie własne

Podsumowując, aplikacja oferuje rozbudowany zestaw funkcjonalności, jednak jej złożoność przekłada się na wysoki próg wejścia dla nowych użytkowników oraz ograniczone wsparcie wdrożeniowe.

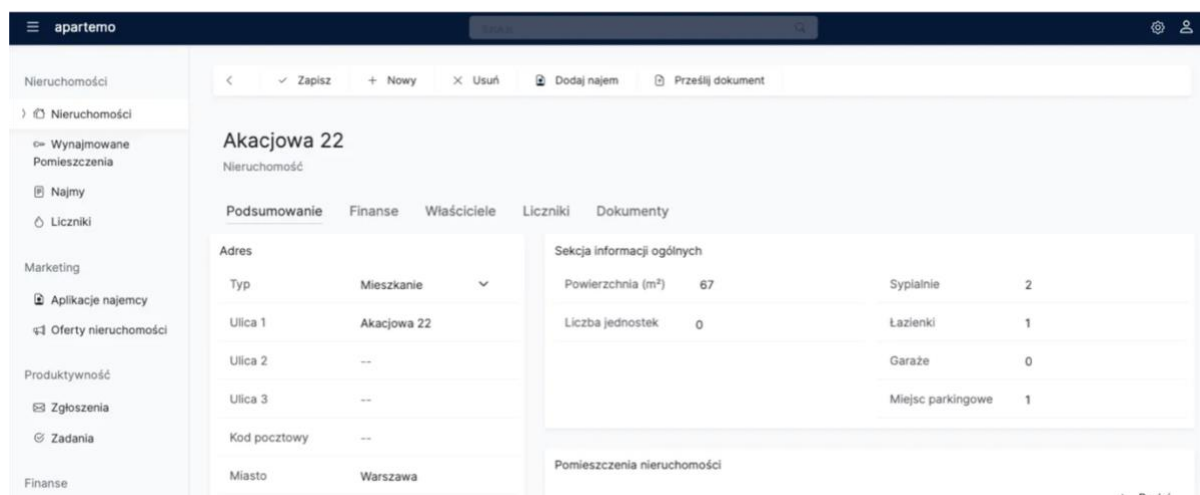
2.2. Aplikacja „Apartemo”

Apartemo [7] jest systemem zarządzania najmem skupionym na procesie operacyjnym wynajmu, w szczególności obsłudze umów, płatności oraz komunikacji z najemcami. Posiada portal najemcy, jednak pełni on rolę pomocniczą w stosunku do panelu zarządcy.

System ten, w porównaniu do poprzedniego rozwiązania, cechuje się znacznie wyższym poziomem intuicyjności oraz orientacji na użytkownika końcowego. Użytkownik nie musi rozumieć technicznych aspektów działania aplikacji, aby sprawnie z niej korzystać. Interfejs jest bardziej nowoczesny niż w przypadku cięższych systemów klasyfikowanych jako ERP²¹, co ułatwia jego wdrożenie i użytkowanie.

Jednocześnie posiada parę ograniczeń funkcjonalnych. Brakuje w nim m.in. modułów finansowych oraz rozbudowanej analityki, co ogranicza jego zastosowanie w większych organizacjach. Dodatkowo w przypadku tej aplikacji portal najemcy nie stanowi głównego elementu produktu, lecz jedynie rozszerzenie systemu zarządcy, co stawia doświadczenia użytkownika końcowego w drugim rzędzie.

²¹ Enterprise Resource Planning - z. ang. zintegrowany system informatyczny służący do zarządzania zasobami firmy



Rysunek 3: Zrzut ekranu Panelu nieruchomości na platformie Apartemo, Źródło: [7]

2.3. Aplikacja „FlatFlow”

Ten system [8] jest małą aplikacją SaaS²² przeznaczoną głównie dla indywidualnych właścicieli nieruchomości oraz małych portfeli mieszkaniowych. System umożliwia podstawowe zarządzanie najmem, ewidencje lokali najemców oraz płatności.

Zaletą tej aplikacji jest prostota, ponieważ nie znajdziemy tam wielu funkcjonalności na wyrost, a dodatkowo posiada intuicyjny interfejs dostosowany do użytkowników nieposiadających doświadczenia w systemach zarządzania nieruchomością.

Ograniczenia tego systemu wynikają przede wszystkim z jego uproszczonej konstrukcji. Brakuje w nim wielu automatyzacji procesów, integracji z systemami zewnętrznymi jak i funkcji analitycznych. Platforma nie posiada też na ten moment aplikacji dla najemcy, ponieważ widnieje taka informacja na ich stronie internetowej. W efekcie obecnie ten system nie jest skalowalny i nie nadaje się dla dużych organizacji zajmujących się najmem.

2.4. Aplikacja „RentFlow”

RentFlow [9] w odróżnieniu od powyższych jest platformą udostępnioną na rynek Europejski, ukierunkowaną na automatyzację procesów, zarządzanie najmem oraz wsparcie operacyjne dla firm zarządzających nieruchomościami. System koncentruje się na workflow²³, raportowaniu oraz usprawnieniu procesów zewnętrznych.

Zaletą systemu jest wysoki poziom automatyzacji, który pozwala na manualne dostosowanie procesów do własnych potrzeb w trakcie trwania jego cyklu. Dodatkowo system

²² Software as a Service - z. ang model dostarczania oprogramowania systemu, bez potrzeby instalacji na własnym serwerze

²³ Workflow - uporządkowany ciąg procesów prowadzących do wykonania określonego zadania, schemat pracy w organizacji lub systemie

oferuje rozbudowane funkcje raportowe, które wspierają analizę operacyjną i podejmowanie decyzji biznesowych.

Jednocześnie tak jak w powyższych platformach ten system charakteryzuje się ograniczonym naciskiem na użytkownika końcowego, ponieważ jest zaprojektowany głównie z perspektywy zarządcy. Interfejs może być mniej intuicyjny dla użytkownika mało technicznego oraz z małym doświadczeniem w najmach nieruchomości. Natomiast można zauważyć, że ta platforma jest skierowana do firm, które posiadają wiele mieszkań i oprócz samego zarządzania najmami potrzebują większej operacyjności związanej z innymi dziedzinami niż najem.

2.5. Aplikacja „Buildium”

Jest to system działający w modelu SaaS [10], szeroko i znacznie wykorzystywanym przez zarządców nieruchomości, zarówno indywidulanych, jak i firmy zarządzające portfelami mieszkań na rynku Amerykańskim. Platforma umożliwia zarządzanie umowami najmu, płatnościami, księgowością, zgłoszeniami serwisowymi oraz komunikacją z najemcami. System oferuje również portal najemcy, który pozwala na dokonywanie płatności, składanie wniosków serwisowych oraz dostęp do dokumentów.

Zaletą tej platformy jest duże urozmaicenie operacji oraz wiele integracji różnych funkcji w jednym środowisku, co umożliwia zarządzanie całym cyklem życia najmu. System oferuje również funkcje związane z pozyskiwaniem najemców oraz ich weryfikację, co bardzo wyróżnia tę platformę na rynku.

Pomimo szerokiej funkcjonalności, system charakteryzuje się stosunkowo dużą złożonością, co może utrudniać wdrożenie nowych użytkowników, szczególnie dla mniejszych właścicieli nieruchomości. Podobnie jak powyższe systemy, tutaj też interfejs oraz logika są głównie budowane z myślą o zarządcy. W efekcie czego portal najemcy pełni funkcje pomocniczą, a nie jako główny moduł systemu. Dużą wadą tego systemu jest też to, że koszty korzystania z platformy rosną wraz ze skalą portfela nieruchomości, co może stanowić barierę dla mniejszych użytkowników.

2.6. Podsumowanie

Przeprowadzona analiza systemów zarządzania najmem, obejmująca zarówno rozwiązania lokalne jak i rynek Europejski oraz Amerykański, a także duże systemy Enterprise²⁴, wskazuje na wyraźne zróżnicowanie rynku pod względem, skali oraz podejścia projektowego. Systemy klasyczne oferują szeroki zakres funkcji operacyjnych, obejmujących zarządzanie umowami, finansami, komunikacją oraz utrzymaniem nieruchomości. Ich główną zaletą jest duża liczba możliwości operacji, jednak oznacza to znaczną złożoność interfejsu oraz wysoki próg wejścia dla użytkowników. Z kolei systemy mniejsze, skierowane do indywidualnych właścicieli, oferują większą intuicyjność i łatwość wdrożenia, jednak kosztem bardzo ograniczonej funkcjonalności oraz braku skalowalności.

²⁴ System Enterprise - rozbudowany system informatyczny przeznaczony do obsługi procesów w dużych organizacjach, integrujący duże obszary działalności

W analizie uwzględniono również nowsze rozwiązania, które próbują zwiększyć rolę najemcy poprzez portale użytkownika oraz aplikacje mobilne. Mimo tego nawet w bardziej rozwiniętych systemach, funkcjonalność najemcy ogranicza się głównie do obsługi płatności, zgłoszeń oraz dostępu do dokumentów.

Kluczowym wnioskiem jest fakt, że większość analizowanych systemów, uwzględnionych w tej pracy jak i również nie uwzględnionych, została zaprojektowana pod kątem zarządcy nieruchomości, a nie użytkownika końcowego. Znaczna część tych platform jest również skierowana do dużych firm, które oprócz najmów potrzebują więcej funkcjonalności związanych ze sprzedażą, marketingiem czy zarządzaniem ludźmi. Sam najemca w tych systemach pełni rolę wtórną i nie jest traktowany jako pełnoprawny użytkownik produktu.

Ważnym elementem, który charakteryzuje powyższe systemy jest to, że nie wszystkie z nich posiadają pełny cykl życia najmu. Nie jesteśmy w stanie w nich procesować i śledzić statusów na jakich znajduje się najem w danej chwili. W wielu przypadkach służą one jako obieg dokumentów, raporty, księgowość oraz odpowiadają za pilnowanie terminów jakie znajdują się w najmie, bez dawania pełnej kontroli nad procesem oraz możliwości podejmowania decyzji w danym momencie.

Na podstawie przeprowadzonej analizy można zidentyfikować istotną lukę rynkową. Brakuje rozwiązań, które integrowałyby funkcje zarządzania najmem z podejściem zorientowanym na użytkownika końcowego. Taki system powinien oferować spójne środowisko obsługi całego cyklu życia najmu, umożliwiać śledzenie jego statusu oraz podejmowanie decyzji w kluczowych momentach, przy jednoczesnej pełnej automatyzacji czynności powtarzalnych.

3. Metodologia wytwarzania oprogramowania

Realizacja projektu informatycznego wymaga zastosowania odpowiednio dobranej metodologii wytwarzania oprogramowania, przed przystąpieniem do fazy implementacyjnej, która pozwala na uporządkowanie procesu projektowego oraz zwiększenie efektywności pracy zespołu. Wybór metodyki uzależniony jest od wielu czynników, takich jak wielkość zespołu, charakter projektu czy jego złożoność. Ze względu na dwuosobowy skład zespołu projektowego oraz iteracyjny charakter realizowanych prac, zdecydowano się na zastosowanie uproszczonego podejścia opartego na metodyce Scrum. Wybrano ten model ze względu na potrzebę bieżącej weryfikacji postępów prac. Proces wytwarzania oprogramowania był dodatkowo wspierany przez narzędzie do zarządzania zadaniami Jira oraz praktyki przeglądu kodu [11] .

3.1. Scrum

Scrum [12] jest jedną z metodyk zwinnych stosowanych w procesie wytwarzania oprogramowania, oparta na podejściu iteracyjnym i przyrostowym. Zakłada on podział pracy na cykle, zwane sprintami, w ramach których realizowane są kolejne funkcjonalności systemu. Każdy sprint kończy się dostarczeniem działającej części systemu, co umożliwia bieżącą ocenę postępów prac. W klasycznym podejściu definiuje określone role, takie jak Scrum Master²⁵, Product Owner²⁶ oraz zespół deweloperski, a także zestaw wydarzeń i artefaktów wspierających zarządzanie projektem. Do najważniejszych elementów należą m.in. backlog²⁷ produktu, planowanie sprintu oraz przegląd wykonanych prac.

W ramach realizowanego projektu zastosowano uproszczoną wersję metodyki Scrum, dostosowaną do dwuosobowego zespołu projektowego. Prace były organizowane w sposób iteracyjny, a zadania realizowano w krótkich cyklach, co umożliwiało stopniowe rozwijanie systemu oraz bieżącą weryfikację jego funkcjonalności.

Zastosowanie podejścia Scrum pozwoliło na elastyczne reagowanie na zmieniające się wymagania oraz efektywne zarządzanie zakresem projektu, co było szczególnie istotne w przypadku niewielkiego zespołu i dynamicznego charakteru prac.

3.2. Jira

Platforma [13] służąca do zarządzania projektami oraz wspomagania procesu wytwarzania oprogramowania. Umożliwia ona tworzenie i organizowanie zadań, monitorowanie postępu prac oraz koordynację działań zespołu projektowego.

W realizowanym projekcie narzędzie zostało wykorzystane do zarządzania zadaniami oraz planowania prac. Utworzona została lista zadań i funkcjonalności przewidzianych do

²⁵ Osoba odpowiedzialna za maksymalizację wartości produktu oraz zarządzanie backlogiem produktu w metodyce Scrum. Reprezentuje interesariuszy i określa priorytety zadań

²⁶ Osoba wspierająca zespół Scrum w prawidłowym stosowaniu zasad Scrum, usuwaniu przeszkód oraz usprawnianiu współpracy w zespole

²⁷ Uporządkowana lista zadań, wymagań i funkcjonalności do wykonania w projekcie, aktualizowana na bieżąco w trakcie realizacji produktu

realizacji, natomiast jeśli pojawiały się nowe pomysły w trakcie trwania projektu to były tworzone na bieżąco w narzędziu. Poszczególne zadania trafiały od członków zespołu oraz były realizowane w kolejnych etapach prac.

Do wizualizacji postępu wykorzystano tablice Kanban²⁸, która umożliwiała śledzenie statusu poszczególnych zadań. Zadania przechodziły przez kolejne etapy, takie jak „To do”, „In progress”, „In review” oraz „Done”, co pozwalało na bieżącą kontrolę stanu projektu.

Zastosowanie narzędzia Jira pozwoliło na uporządkowanie procesu realizacji projektu, zwiększenie przejrzystości prac oraz efektywne zarządzanie zadaniami w ramach niewielkiego zespołu projektowego.

3.3. Code review (weryfikacja jakości kodu)

Code review inaczej (przegląd kodu) jest praktyką stosowaną zawsze w programowaniu komercyjnym, która ma na celu systematyczne przeglądanie zmian w kodzie źródłowym przez innego członka zespołu przed ich dodaniem do głównej gałęzi repozytorium. Celami tego procesu są: wykrywanie błędów, poprawa jakości kodu, transfer wiedzy między członkami zespołu [14] .

W projekcie w tym celu został użyty mechanizm Merge Request²⁹, który jest możliwy do użycia na platformie Gitlab. Każda z nowych funkcjonalności była rozwijana na oddzielnej gałęzi feature³⁰ i po zakończeniu pracy zgłaszana do przeglądu. Drugi członek zespołu weryfikował poprawność implementacji, zgodność z przyjętymi konwencjami kodowania i testów. W momencie akceptacji zmiany były scalane z gałęzią główną.

Przeglądy kodu w obecnym projekcie były realizowane z wykorzystaniem systemu kontroli wersji GIT³¹, co umożliwiało śledzenie wprowadzanych zmian. Dzięki zastosowaniu takiej metodyki możliwe było szybkie wykrywanie potencjalnych błędów oraz wprowadzanie usprawnień na wczesnym etapie rozwoju danej funkcjonalności systemu.

²⁸ Narzędzie wizualnego zarządzania pracą, które przedstawia zadania w podziale na etapy procesu

²⁹ Wprowadzenie zmian w kodzie źródłowym poprzez połączenie jednej gałęzi repozytorium z inną, zwykle po wcześniejszym przeglądzie i akceptacji zmian przez zespół

³⁰ Wydzielona gałąź w systemie kontroli wersji, tworzona w celu implementacji konkretnej funkcjonalności lub poprawki bez wpływu na główną wersję kodu projektu

³¹ Rozproszony system kontroli wersji służący do śledzenia zmian w plikach oraz współpracy wielu osób nad projektem

4. Propozycja systemu do zarządzania najmem krótko oraz długoterminowym

Przeprowadzając badanie rynku najmów nieruchomości, udało się zdefiniować kluczowe aspekty jakie charakteryzują już istniejące systemy. Pozwoliło to na wyodrębnienie ich wad i zalet, co przyczyniło się do stworzenia prototypu nowego rozwiązania, który dostosowuje się do obecnego rynku oraz implementuje nowe funkcjonalności w inny, bardziej odpowiedni i intuicyjny dla użytkownika końcowego sposób. Przed przystąpieniem do prac projektowych konieczne było zidentyfikowanie niszy rynkowej oraz określenie obszarów, w których system mógłby realnie usprawnić codzienną pracę zarządców. Dopiero na tej podstawie możliwe było opracowanie rozwiązania odciążającego ich od rutynowych czynności administracyjnych. Dlatego powstała wizja opisywanego systemu, która udowadnia sens oraz możliwości jakie niesie ze sobą to rozwiązanie.

4.1. Wizja systemu

W obecnych czasach, gdzie następuje szybki rozwój branży najmów długoterminowych i krótkoterminowych, aby efektywnie zarządzać wieloma nieruchomościami i być konkurencyjnym, potrzebne jest oprogramowanie. Zarządzanie portfelem nieruchomości na wynajem wymaga ciągłego nadzoru nad wieloma obszarami jednocześnie: dostępność lokali, dokumentacją prawną, finansami, personelem serwisowym i komunikacją z najemcami. Właściciele zarządzający wieloma lokalami, często w różnych miastach mają problem z synchronizacją informacji z różnych źródeł. Dostępne na rynku rozwiązania bardzo często nie nadążają nad szybkim rozwojem branży najmów, nie mają możliwości obsługi obu modeli najmu w ramach jednej aplikacji. Wiele z nich posiada tylko platformy skupione bezpośrednio na zarządcy, pomijając w tym wszystkim najemcę, który nie otrzymuje systemu i tym samym nie ma możliwość podglądu swojego najmu. Zauważamy też, że te systemy obecne dziś na rynku są głównie przeznaczone do obiegu dokumentów oraz pilnowania terminów nałożonych na starcie najmu, co przyczynia się do większego ryzyka błędów w trakcie ze względu na to, że zarządca nie ma możliwości sterowania najmem, ani podejmowania decyzji w trakcie jego cyklu. Opisywany system odpowiada na powyższe problemy poprzez automatyzację powtarzalnych czynności administracyjnych oraz wsparcie codziennych procesów biznesowych. Kluczową cechą rozwiązania jest przyznanie zarówno zarządcy, jak i najemcy pełnej kontroli nad przebiegiem najmu na każdym etapie jego cyklu, co jest w dzisiejszych czasach bardzo ważne.

System posiada dwie główne domeny użytkowników. Zarządca dostaje kompletne narzędzia do wprowadzania pełnego cyklu najmu, zaczynając od rejestracji lokalu i weryfikacji firmy po generowanie umów, rozliczenia miesięcznie z fakturowaniem, aż po zakończenie najmu i archiwizację dokumentacji. Natomiast najemca otrzymuje dostęp do dedykowanego portalu, który umożliwia podgląd aktywnych umów, historii opłat, statusu najmu, statusu płatności, dokumentów oraz odczytywania zużycia mediów.

Najważniejsze obszary funkcjonalne systemu obejmują zarządzanie lokalami, umowami najmu, automatyczne generowanie cyklicznych należności, weryfikacje podmiotów

gospodarczych za pośrednictwem publicznych rejestrów GUS³², obsługę płatności z wystawianiem faktur, system powiadomień e-mail oraz rejestrację odczytów mierników mediów. Bezpieczeństwo aplikacji jest realizowane przez własny serwis autoryzacyjny z tokenami JWT weryfikowanymi certyfikowanym kluczem.

System został stworzony jako aplikacja webowa, która jest dostępna z przeglądarki internetowej, zbudowana została w oparciu o architekturę mikroservisową. Zostały zaimplementowane serwisy BFF, co oznacza, że każda aplikacja frontendowa komunikuje się poprzez dedykowany serwis, który jest łącznikiem z serwisami wewnętrznymi, gdzie znajduje się logika biznesowa systemu. Serwisy komunikują się synchronicznie za pośrednictwem REST API oraz asynchronicznie przez broker wiadomości RabbitMQ. Całość środowiska została skonteneryzowana przy użyciu Dockera i orkiestrowana za pomocą dystrybucji Kubernetes, a prototyp systemu został wdrożony na wirtualnej maszynie w sieci uczelnianej

4.2. Specyfikacja wymagań funkcjonalnych

Przed rozpoczęciem prac nad aplikacją potrzebne jest określenie wymagań systemu, w kontekście zachowania, logiki i specyfikacji działania. Wcześniejsze zaprojektowanie takich zależności bez implementacji pomaga wykryć w początkowej fazie projektu problemy jakie może nieść ze sobą dane rozwiązanie lub zauważyć, że jakaś część implementacji może być niepotrzebna na początku. Odnosząc się do projektu, któremu poświęcona jest ta praca, wymagania zostały stworzone na podstawie analizy procesów biznesowych występujących w pracy zarządców nieruchomości. Badanie wykazało, że przeniesienie wielu rutynowych czynności do centralnego systemu, przy zachowaniu spójnej i poprawnej implementacji, stanowi istotne usprawnienie zarówno dla zarządców, jak i najemców.

4.2.1 Zrządzanie lokalami

Moduł zarządzania lokalami stanowi centralny rejestr oraz portfel nieruchomości wynajmującego. Każdy lokal funkcjonuje w systemie jako samodzielny obiekt z pełnym zestawem danych opisowych, lokalizacyjnych i wizualnych, który może być w dowolnym momencie przeglądany, edytowany lub usunięty. Każda nieruchomość posiada następujące dane:

- nazwę,
- opis słowny,
- typ lokalu,
- aktualny status (np. dostępny, wynajęty),
- powierzchnię w metrach kwadratowych,
- liczbę pokoi,
- numer piętra,
- numer katastralny/numer księgi wieczystej,

³² Główny Urząd Statystyczny - Centralny organ administracji publicznej w Polsce odpowiedzialny za gromadzenie, opracowywanie i publikowanie danych statystycznych dotyczących m.in. gospodarki, społeczeństwa i demografii kraju

- Adres, w którego skład wchodzi ulica, numer mieszkania, kod pocztowy, miasto oraz kraj,
- współrzędne geograficzne (szerokość i długość) - interaktywnie wskazywane na mapie.

Formularz tworzenia nowego lokalu oraz formularz edycji istniejącego mają identyczną strukturę. Użytkownik wypełnia dane opisowe nieruchomości, a lokalizację wskazuje bezpośrednio na interaktywnej mapie osadzonej w formularzu - kliknięcie wybranego punktu na mapie automatycznie uzupełnia współrzędne geograficzne, eliminując potrzebę ręcznego wpisywania danych, natomiast jeśli będzie taka potrzeba manualne wpisanie wartości jest również możliwe.

Do każdego lokalu można dołączyć dowolną liczbę zdjęć w formatach JPEG³³, PNG³⁴ lub WebP³⁵. System weryfikuje format i rozmiar każdego przesyłanego pliku wynoszący 70MB na request oraz 7MB na jedno zdjęcie. Zdjęcia lokalu tworzą galerię, w której każde zdjęcie posiada swoją pozycję wyświetlania oraz oznaczenie. Zdjęcia można dodawać przy tworzeniu lub edycji lokalu, a także usuwać wybrane z nich w dowolnym momencie bez konieczności ponownego przesyłania całej galerii.

Usunięcie lokalu jest operacją trwałą. System usuwa najpierw wszystkie powiązane zdjęcia, a następnie sam rekord lokalu, nie pozostawiając danych bez relacji. Użytkownik widzi wyłącznie lokale przypisane do jego organizacji - system automatycznie filtruje dane na podstawie tożsamości zalogowanego użytkownika i jego przynależności. Widok listy prezentuje wszystkie lokale wraz z ich zdjęciami, natomiast widok szczegółów pojedynczego lokalu zawiera pełen zestaw atrybutów opisowych, galerię zdjęć, lokalizację na mapie oraz powiązane odczyty liczników mediów.

Z poziomu szczegółów lokalu użytkownik ma wgląd w historię i aktualny stan najmu danej nieruchomości. Widoczne są daty zajętości lokalu, co umożliwia planowanie nowych umów bez ryzyka nakładania się terminów. posiada zestaw atrybutów: adres, metraż, liczbę pokoju, status, opis, nazwę, numer katastralny, współrzędne.

4.2.2 Zarządzanie kontami i uprawnieniami

System opiera się na modelu kontroli dostępu opartym na rolach RBAC³⁶, w którym każdy użytkownik posiada przypisany zestaw uprawnień determinujący, do których funkcji systemu ma dostęp. Osoba zakładająca konto główne wynajmującego automatycznie otrzymuje pełen zestaw uprawnień i staje się administratorem swojej organizacji w systemie. Administrator może tworzyć konta dla swoich pracowników i współpracowników, samodzielnie dobierając im role odpowiednie do zakresu ich obowiązków. Nowo utworzone konto pracownika jest nieaktywne – jednakże pracownik otrzymuje e-mail z linkiem aktywacyjnym i może samodzielnie ustawić swoje hasło przed pierwszym logowaniem. W systemie dostępne są następujące role:

³³ JPEG (norma ISO/IEC 10918-1) - strata kompresja obrazów rastrowych

³⁴ Portable Network Graphics - bezstratny format grafiki rastrowej, obsługujący przezroczystość i wysoką jakość obrazu

³⁵ Nowoczesny format obrazu stworzony przez Google, oferujący kompresję stratań i bezstratną przy mniejszym rozmiarze plików niż PNG lub JPEG

³⁶ RBAC (Role-Based Access Control) - model kontroli dostępu, w którym uprawnienia przypisywane są do ról, a nie bezpośrednio do użytkowników. Użytkownik uzyskuje dostęp do zasobów poprzez przypisaną mu rolę. Standard: NIST SP 800-162

- **LANDLORD_ADMIN_ROLE** - rola administratora. Automatycznie otrzymuje ją użytkownik, który zakłada główne konto zarządcy oraz może zostać nadana każdemu użytkownikowi w późniejszym czasie. Użytkownik może zarządzać kontami innych pracowników: tworzyć nowe konta, przeglądać listę wszystkich użytkowników przypisanych do organizacji oraz modyfikować ich uprawnienia,
- **LANDLORD_MANAGER_ROLE** - rola manager. Użytkownik ma dostęp do kluczowych operacji zarządczych: tworzenia, edytowania i usuwania lokali, zawierania nowych umów najmu, konfigurowania konta bankowego wynajmującego oraz przeglądania odczytów liczników mediów,
- **LANDLORD_FINANCE_ROLE** - rola finansowa. Użytkownik ma dostęp do modułów rozliczeniowych systemu: historii płatności, harmonogramów opłat oraz dokumentów finansowych powiązanych z umowami najmu,
- **LANDLORD_TENANT_RELATIONS_ROLE** - rola ds. relacji z najemcami. Użytkownik może zarządzać danymi najemców oraz obsługiwać kwestie związane z ich kontami i komunikacją,
- **LANDLORD_USER_ROLE** - rola podstawowego użytkownika domeny zarządcy. Zapewnia dostęp do przeglądania listy umów najmu oraz podstawowych funkcji operacyjnych systemu,
- **LANDLORD_READ_ONLY_ROLE** - rola tylko do odczytu. Użytkownik może przeglądać dane lokali i umów bez możliwości dokonywania jakichkolwiek zmian. Rola przeznaczona dla osób, które potrzebują wglądu w dane, ale nie powinny ich modyfikować,
- **TENANT_USER_ROLE** - rola nadawana najemcom posiadającym konta na platformie. Pozwala na operacje po stronie aplikacji najemcy, w tym przeglądanie swoich zawartych najmów, opłaty, pobieranie dokumentów.

Jeden użytkownik może posiadać kilka ról jednocześnie, co pozwala elastycznie dopasować zakres dostępu do rzeczywistego zakresu obowiązków pracownika. Administrator może w dowolnym momencie zmienić przypisane role, odbierając lub nadając uprawnienia bez konieczności usuwania i zakładania konta od nowa.

4.2.3 Obsługa umów najmu oraz ich procesowanie

Moduł najmu jest centralnym elementem systemu, który łączy wynajmującego, najemcę i lokal w jeden dokument określający warunki współpracy. Wynajmujący tworzy nową umowę, wskazując konkretny lokal oraz wypełniając dane najemcy - jego dane kontaktowe i identyfikacyjne. Jednocześnie definiuje warunki finansowe: ustala, jakie opłaty najemca będzie regulował co miesiąc (np. czynsz, media, kaucja), ich wysokość oraz termin płatności. Podaje też daty obowiązywania umowy - z określonym końcem lub bezterminowo. W momencie zapisania umowy system automatycznie wysyła do najemcy wiadomość e-mail z zaproszeniem do portalu, gdzie będzie mógł śledzić swoje płatności i dokumenty. Wynajmujący ma w każdej chwili dostęp do listy wszystkich swoich umów wraz z podstawowymi informacjami o każdej z nich: numerem umowy, statusem, danymi najemcy i powiązanym lokalem. Przed zawarciem nowej umowy system umożliwia też sprawdzenie, w jakich terminach dany lokal jest już zajęty, co pozwala uniknąć nakładania się okresów najmu.

4.2.4 System powiadomień

System powiadomień działa jako niezależny komponent, który nasłuchuje na zdarzenia zachodzące w innych częściach platformy i w odpowiedzi na nie wysyła wiadomości e-mail do zainteresowanych stron. Komunikacja między serwisami odbywa się za pośrednictwem kolejki wiadomości RabbitMQ³⁷, co zapewnia, że żadne powiadomienie nie zostanie pominięte nawet w przypadku chwilowej niedostępności serwisu pocztowego. Każda wiadomość e-mail jest generowana na podstawie dedykowanego szablonu HTML z uzupełnionymi danymi konkretnymi dla danego zdarzenia. System obsługuje następujące zdarzenia:

- Aktywacja konta najemcy - w momencie zawarcia nowej umowy najmu, gdy najemca jest nowym użytkownikiem platformy, system wysyła do niego wiadomość powitalną zawierającą indywidualny link aktywacyjny, który umożliwia mu ustawienie hasła i uzyskanie dostępu do portalu,
- Aktywacja konta użytkownika przypisanego do panelu zarządcy - gdy administrator tworzy konto nowego pracownika, ten otrzymuje e-mail z linkiem umożliwiającym aktywację konta i ustawienie hasła do panelu,
- Weryfikacja głównego konta zarządcy - podczas zakładania konta głównego użytkownik, któremu zostaje nadane automatycznie uprawnienie administratora, aby móc operować na koncie musi zweryfikować najpierw swój email,
- Zawarcie umowy najmu - po pomyślnym utworzeniu umowy do najemcy wysyłane jest potwierdzenie z informacją o szczegółach podpisanego najmu,
- Rozpoczęcie najmu - w dniu, w którym najem oficjalnie się rozpoczyna, najemca otrzymuje powiadomienie z potwierdzeniem startu umowy wraz z danymi lokalu,
- Miesięczne potwierdzenie płatności - po zaksięgowaniu każdej cyklicznej płatności czynszowej najemca otrzymuje e-mail z potwierdzeniem za dany okres rozliczeniowy,
- Zakończenie najmu - w dniu zakończenia najmu, do najemcy oraz zarządcy, który opiekuje się danym najmem zostaje wysłany mail powiadamiający o zakończeniu najmu i możliwości rozpoczęcia procedury kończącej proces,
- Przypomnienie o zbliżającym się końcu umowy - na 7 dni przed datą zakończenia umowy terminowej system automatycznie wysyła do najemcy przypomnienie o kończącym się najmie.

Zakończenie umowy - w dniu wygaśnięcia umowy powiadomienie o jej zakończeniu jest wysyłane jednocześnie do najemcy oraz do wynajmującego.

Powiadomienia SMS nie zostały zaimplementowane w bieżącej wersji systemu - platforma obsługuje wyłącznie komunikację e-mailową.

4.2.5 System pomiarów liczników w mieszkaniach

System automatycznie pobiera i aktualizuje odczyty liczników dla każdej nieruchomości w regularnych odstępach czasu - co 30 minut. Dla każdego lokalu śledzone są pięć rodzajów mediów: energia elektryczna, ogrzewanie, zimna woda, ciepła woda oraz gaz. Każde mieszkanie posiada osobny licznik z przypisaną jednostką miary - kilowatogodziny (kWh)³⁸ dla energii i ogrzewania, metry sześciennie (m³)³⁹ dla wody i gazu. Przy pierwszym pobraniu

³⁷ Mechanizm zdarzeń implementujący protokół AMQP (Advanced Message Queuing Protocol). Wzorzec kolejki wiadomości gwarantuje trwałość zdarzeń i odłączenie producenta od konsumenta

³⁸ Kilowatogodzina (kWh) - jednostka energii elektrycznej równa 3,6 MJ, stosowana do rozliczeń energetycznych

³⁹ Metr sześcienny - jednostka SI używana do rozliczania wody i gazu

danych dla danego lokalu system inicjalizuje wszystkie pięć liczników. W kolejnych cyklach wartości są aktualizowane przez dodanie przyrostu zużycia od poprzedniego odczytu, co odzwierciedla faktyczne, narastające wskazanie licznika. W docelowym środowisku produkcyjnym dane byłyby pobierane w każdym cyklu z zewnętrznego API dostawcy mediów. Na potrzeby projektu zaimplementowano zaślepkę, która symuluje realistyczne zużycie, generując losowy przyrost wartości dla każdego licznika w każdym cyklu. Wynajmujący ma dostęp do bieżących odczytów wszystkich liczników danego lokalu z poziomu panelu zarządzania, co pozwala na bieżące monitorowanie zużycia mediów przez zarządcę.

4.2.6 Rejestracja i logowanie

System powinien umożliwiać założenie konta i weryfikację danych osoby, która będzie wynajmować nieruchomości. Aplikacja powinna obsługiwać dwie różne grupy użytkowników - wynajmujących i najemców. Każda z nich trafia do systemu w inny sposób. Wynajmujący rejestruje się samodzielnie, wypełniając formularz ze swoimi danymi: nazwę wyświetlaną, typem konta (osoba fizyczna lub firma), adresem, danymi kontaktowymi oraz hasłem. Po pomyślnej rejestracji jest od razu zalogowany i może korzystać z systemu. Wynajmujący może też dodawać do swojego konta kolejnych użytkowników - np. pracowników lub współpracowników, przydzielając im odpowiednie uprawnienia i role w systemie. Najemca nie rejestruje się sam. Jego konto powstaje automatycznie w momencie, gdy najemca jest nowy na platformie oraz wynajmujący tworzy umowę najmu i podaje jego dane. System wysyła wówczas na adres e-mail najemcy wiadomość z unikalnym linkiem aktywacyjnym. Klikając link, najemca potwierdza swoją tożsamość i ustawia własne hasło. Od tej chwili może logować się do portalu najemcy tak samo jak każdy użytkownik - podając adres e-mail i hasło.

Logowanie przebiega tak samo dla obu grup: użytkownik podaje swój adres e-mail i hasło, a system w odpowiedzi wydaje token dostępu⁴⁰, który autoryzuje wszystkie kolejne działania w aplikacji. Token zawiera informację o tym, kim jest użytkownik i jakie ma uprawnienia, dzięki czemu system wie, do których funkcji dana osoba ma dostęp.

4.2.7 Cykliczne generowanie należności

System powinien generować i wysyłać należności, klientom, którzy mają aktywne najmy. W momencie zawarcia umowy najmu wynajmujący definiuje harmonogram opłat - ustala, jakie należności najemca będzie regulował co miesiąc (np. czynsz), ich wysokość oraz dzień miesiąca, w którym upływa termin płatności. System zapamiętuje ten schemat i na jego podstawie automatycznie generuje kolejne należności w każdym kolejnym okresie rozliczeniowym, bez żadnej dodatkowej interwencji ze strony wynajmującego. Pierwsza należność jest tworzona od razu przy podpisaniu umowy. Jeśli najem zaczyna się w środku miesiąca, system proporcjonalnie pomniejsza kwotę pierwszego czynszu - najemca płaci tylko za faktyczną liczbę dni, które pozostały do końca miesiąca. Po zaksięgowaniu każdej płatności system automatycznie generuje fakturę dostępną do wglądu w panelu najemcy jak i zarządcy. Jednocześnie tworzy kolejną należność na następny okres, dzięki czemu cykl rozliczeniowy trwa nieprzerwanie przez cały czas obowiązywania umowy.

4.2.8 Dostępność nieruchomości

⁴⁰ Skrócony dowód tożsamości wydawany przez serwis autoryzacyjny, najczęściej w formacie JSON. Niesie zakodowane informacje pozwalające weryfikować tożsamość i uprawnienia bez odpytywania bazy

Zarządca przed utworzeniem najmu ma dostęp do kalendarza danego mieszkania i tym samym może sprawdzić dostępność lokalu w wybranych przez siebie dniach. Umożliwia to kontrolka kalendarza, która ma zablokowane niedostępne terminy. Ten kalendarz automatycznie pobiera z systemu wszystkie okresy, w których lokal jest już zajęty na podstawie istniejących umów, i blokuje te daty jako niedostępne. Dzięki temu wynajmujący od razu widzi, które dni są wolne, a próba wybrania zajętego terminu jest niemożliwa - kalendarz jak i system wewnątrz nie pozwala na zaznaczenie zablokowanych dat. Rozwiązanie to eliminuje ryzyko omyłkowego nakładania się umów dla tego samego lokalu.

4.2.9 Generowanie dokumentów PDF

System automatycznie generuje dokumenty PDF⁴¹ dla kluczowych zdarzeń w cyklu życia umowy najmu, bez konieczności jakiegokolwiek interwencji ze strony użytkownika. Dokumenty są tworzone asynchronicznie w tle i przechowywane w dedykowanym serwerze obiektowym, skąd mogą być w dowolnym momencie pobrane przez obie strony - zarówno wynajmującego, jak i najemcę. System generuje dwa rodzaje dokumentów:

- Umowa najmu jest generowana automatycznie zaraz po zawarciu umowy. Dokument zawiera pełne dane obu stron - wynajmującego (nazwa, adres, dane identyfikacyjne, e-mail, telefon) oraz najemcy (imię i nazwisko lub nazwa firmy, dane identyfikacyjne, e-mail, telefon) - a także szczegółowy opis wynajmowanej nieruchomości (adres, typ, powierzchnia, liczba pokoi, numer piętra, numer katastralny) i warunki finansowe: numer umowy, daty obowiązywania, wysokość czynszu, kwotę kaucji oraz dzień miesiąca, w którym upływa termin płatności,
- Faktura miesięczna jest generowana automatycznie po każdym potwierdzeniu płatności przez serwis płatności. Dokument zawiera dane wynajmującego i najemcy, numer faktury, datę wystawienia, okres rozliczeniowy (od-do), termin płatności, łączną kwotę oraz rozbieżność na poszczególne pozycje (np. czynsz, media). Dokumenty są przypisane do konkretnej umowy najmu i dostępne do przeglądania w podziale na typ - użytkownik może najpierw wyświetlić listę dostępnych plików dla danej umowy. Pobieranie odbywa się przez strumieniowanie pliku bezpośrednio z serwisu przechowywania, co eliminuje konieczność tymczasowego zapisywania dokumentu po stronie serwera aplikacji.

4.3. Specyfikacja wymagań poza funkcjonalnych

W niniejszym rozdziale przedstawiono wymagania poza funkcjonalne dotyczące projektowanego systemu. W odróżnieniu od wymagań funkcjonalnych opisanych w podrozdziale powyżej, które określają, jakie funkcje system ma realizować, wymagania poza funkcjonalne opisują sposób, w jaki system powinien te funkcje wykonywać oraz jakie cechy jakościowe powinien spełniać.

Wymagania te odnoszą się do wydajności, niezawodności, bezpieczeństwa, skalowalności oraz użyteczności czy łatwości utrzymania systemu. Ich właściwa charakterystyka ma kluczowe znaczenie dla zapewnienia odpowiedniej jakości rozwiązania oraz jego zgodności z oczekiwaniami użytkowników i interesariuszy.

⁴¹ Portable Document Format - format niezależny od platformy, standaryzowany jako: ISO 32000-1:2008

W odróżnieniu od wymagań funkcjonalnych, które są zwykle łatwe do bezpośredniego przetestowania, wymagania poza funkcjonalne często przyjmują formę mierzalnych kryteriów systemu, które muszą być spełnione w określonych warunkach działania. Z tego względu stanowią one istotny element procesu projektowania architektury systemu oraz jego późniejszej implementacji i testowania.

4.3.1 Bezpieczeństwo i autoryzacja

Bezpieczeństwo aplikacji webowych jest jednym z kluczowych aspektów współczesnego tworzenia oprogramowania. W aplikacjach opartych o zależność Spring Security [15] stosuje się mechanizmy pozwalające chronić dane użytkowników, kontrolować dostęp do zasobów oraz zabezpieczać aplikację przed najczęściej występującymi zagrożeniami opisanymi przez organizację OWASP, a szczególnie kieruje się zabezpieczeniem przed dziesięcioma najpopularniejszymi modelami ataków i zagrożeń w aplikacjach internetowych [16].

Bezpieczeństwo systemu, któremu poświęcona jest ta praca również od fundamentów wspierane jest poprzez Spring Security, natomiast samo uwierzytelnienie użytkownika opiera się na standardzie OAuth 2.0⁴² w połączeniu z kluczami uwierzytelniającymi JWT.

Uwierzytelnienie - OAuth 2.0 i JSON Web Token

Uwierzytelnianie użytkowników w systemie opiera się na standardzie OAuth 2.0 w połączeniu z kluczami uwierzytelniającymi wspomnianymi wyżej. Po poprawnym zalogowaniu serwis autoryzacyjny wystawia podpisany klucz, który klient dołącza do nagłówka jako wartość Authorization Bearer⁴³, każdego kolejnego żądania⁴⁴ HTTP. Klucz ma ograniczony czas ważności wynoszący 3600 sekund (1 godzina), po czym wygasa i wymagane jest ponowne zalogowanie.

Klucze uwierzytelniające JWT podpisywane są algorytmem RS256⁴⁵. Serwis autoryzacyjny podpisuje konkretne poświadczenie kluczem prywatnym, natomiast każdy serwis wejściowy weryfikuje podpis przy użyciu klucza publicznego dostępnego poprzez endpoint JWKS⁴⁶. Dzięki temu weryfikacja takiego tokenu odbywa się lokalnie w każdym serwisie bez konieczności odpytywania serwisu autoryzacyjnego przy każdym żądaniu - jest to podejście określane jako weryfikacja bezstanowa.

Jednorazowe klucze tymczasowe

Dla operacji ustawiania hasła przed pierwszym logowaniem użytkownika, system wystawia osobny, tymczasowy klucz z uprawnieniem jedynie do zmiany hasła i skróconym czasem ważności wynoszącym zaledwie 5 minut. Token uprawnia wyłącznie do jednej operacji i nie daje dostępu do żadnej innej części systemu. Po ustawieniu hasła zostaje on zastąpiony tokenem

⁴² OAuth 2.0 - otwarty standard autoryzacji (RFC 6749) umożliwiający aplikacji uzyskanie ograniczonego dostępu do zasobów w imieniu użytkownika bez przekazywania jego danych uwierzytelniających. Definiuje przepływy autoryzacji (ang. grant types) dla różnych scenariuszy użycia.

⁴³ Sposób przekazywania poświadczeń w nagłówku HTTP „Authorization”, w którym używany jest token dostępu poprzedzony słowem „Bearer”

⁴⁴ Strukturalny komunikat protokołu HTTP, którym klient inicjuje komunikację z serwerem, prosząc o zasób lub wykonanie operacji.

⁴⁵ RS256 - asymetryczny schemat kryptograficzny oparty na parze kluczy RSA

⁴⁶ Adres URL zwracający zestaw kluczy publicznych w formacie JSON, używanych do weryfikacji tokenów JWT

sesji głównej. Mechanizm ten wyklucza konieczność stosowania haseł tymczasowych tworzonych przez administratora systemu.

Separacja przestrzeni użytkowników

System wprowadza pojęcie Realm⁴⁷, które tworzy oddzielną przestrzeń dla domeny wynajmujących i najemców. Użytkownik zalogowany w przestrzeni wynajmującego nie może użyć swojego uwierzytelnienia w portalu najemcy, natomiast najemca nie może użyć uwierzytelnienia do zalogowania się w panelu zarządcy. Taka przestań zawarta w uwierzytelnieniu użytkownika jest weryfikowana przez serwisy wejściowe, uniemożliwiając przekroczenie granicy między tymi przestrzeniami.

Kontrola dostępu oparta na rolach RBAC

Autoryzacja w systemie - tak jak wspomniano wyżej w podrozdziale 4.2.2, realizowana jest modelem RBAC. Uprawnienia użytkownika są zawarte w kluczu uwierzytelniającym JWT zwane technicznie jako scope'y (np. **landlord:manager**, **landlord:readonly**, **tenant:user**). Każdy endpoint serwisu wejściowego sprawdza obecność wymaganego scypułu przed użyciem operacji. Podejście to zapewnia zasadę minimalnych uprawnień, które cechuje się tym, że każdy użytkownik ma dostęp tylko do części systemu, których faktycznie potrzebuje do wykonywania swoich obowiązków.

Bezpieczeństwo haseł

Hasła użytkowników przechowywane są wyłącznie w postaci hasza⁴⁸ generowanego algorytmem Bcrypt⁴⁹, który jest odporny na ataki. Hasło w postaci jawnej nigdy nie jest zapisywane i logowanie.

System wymusza politykę haseł: minimalna długość 8 znaków, maksymalna 64 znaki, wymaga obecności co najmniej jednej litery i jednej cyfry oraz jednego znaku specjalnego. Walidacja realizowana jest wyrażeniem regularnym po stronie serwera, co uniemożliwia ominięcie wymagań przez modyfikacje żądania po stronie klienta.

Walidacja danych wejściowych

Wszystkie dane przesyłane przez użytkowników podlegają walidacji po stronie serwera przy użyciu wariatorów predefiniowanych oraz własnych opartych na specyfikacji Ja karta Validation [17] . Walidacja obejmuje obecność wymaganych pól, format danych (np. wyrażenie regularne dla haseł), zakresy liczbowe (współrzędne geograficzne: szerokość -90 do 90, długość -180 do 180) oraz typy i rozmiary przesyłanych plików (zdjęcia: wyłącznie JPEG, PNG, Web). Błędne dane są odrzucane przed dotarciem do logiki biznesowej.

System wykrywa również próby tworzenia duplikatów np. rejestracja wynajmującą z już istniejącym adresem e-mail czy dodanie lokalu z numerem katastralnym, który jest już zajęty. W takich przypadkach zwracany jest odpowiedni kod do użytkownika.

⁴⁷ Przestrzeń tworzona dynamicznie dla konta użytkownika systemu, która określa granice w dostępie tylko do określonej logiki danej domeny, w której znajduje się osoba zalogowana

⁴⁸ Ciąg znaków o stałej długości generowany na podstawie danych wejściowych

⁴⁹ Bcrypt - funkcja hashowania haseł zaprojektowana przez Nielsa Provosa i Davida Mazièresa w 1999 roku, oparta na algorytmie szyfrowania Blowfish

Ochrona przed nieautoryzowanym dostępem do zasobów

Na poziomie aplikacji stosowana jest weryfikacja zasobów, identyfikator wynajmującego lub najemcy odczytywany jest z klucza uwierzytelniającego JWT, a nie z parametrów żądania. Uniemożliwia to scenariusz, w którym zalogowany użytkownik próbuje uzyskać dostęp do danych należących do innej organizacji przez manipulację parametrami URL.

CORS oraz CSRF

Każdy serwis backendowy może być skonfigurowany z listą dozwolonych źródeł, czyli domen, z których przeglądarka może wykonywać zapytania do API. Mechanizm ten działa po stronie przeglądarki, która przy każdym żądaniu cross-origin⁵⁰ wysyła nagłówek Origin i sprawdza, czy serwer w odpowiedzi zwraca odpowiednie nagłówki CORS. Jeśli źródło żądania znajduje się na liście dozwolonych, przeglądarka pozwala aplikacji frontendowej uzyskać dostęp do odpowiedzi. W przeciwnym razie odpowiedź zostaje zablokowana i nie jest dostępna dla kodu JavaScript⁵¹, nawet jeśli samo żądanie zostało wysłane i poprawnie obsłużone przez serwer.

Również w aplikacji z powodu bezstanowości mechanizm ochrony przed atakami CSRF⁵² został wyłączony poprzez konfigurację Spring Security. Oznacza to, że serwer nie wymaga kluczy uwierzytelniających CSRF przy wykonywaniu żądań modyfikujących stan (np. POST, PUT, DELETE).

Decyzja ta jest uzasadniona charakterem aplikacji - w przypadku API wykorzystywanego przez dedykowaną aplikację kliencką lub klientów zewnętrznych, ochrona CSRF jest często zastępowana innymi mechanizmami bezpieczeństwa, takimi jak uwierzytelnianie oparte o tokeny (np. JWT), które są implementowane w opracowanym systemie oraz kontrola dostępu na poziomie API. W takim modelu CSRF nie stanowi typowego wektora ataku, ponieważ przeglądarka nie zarządza automatycznie kluczami autoryzacyjnymi.

4.3.2 Skalowalność systemu

Skalowalność systemu rozpatrywana jest na kilku poziomach, zaczynając od optymalizacji na poziomie poszczególnych serwisów, po mechanizmy w warstwie przetwarzania danych. Istotnym elementem jest również architektura całego systemu, w tym podejście mikroserwisowe, które umożliwia niezależne skalowanie wybranych komponentów w zależności od obciążenia. W praktyce oznacza to możliwość dynamicznego zwiększania liczby instancji usług w odpowiedzi na rosnący ruch.

Niezależne skalowanie serwisów

Największą zaletą architektury mikroserwisowej jest możliwość niezależnego skalowania poszczególnych elementów systemu. Jeśli w systemie pojawi się wąskie gardło, na przykład zwiększony ruch w serwisie płatności w dniu terminu płatności, można uruchomić dodatkowe

⁵⁰ Żądanie HTTP wysyłane z jednej domeny (origin) do innej, np. z aplikacji frontendowej działającej na jednej domenie do API hostowanego na innej

⁵¹ Technologia używana głównie w przeglądarkach internetowych do tworzenia interaktywnych elementów stron oraz w aplikacjach webowych, np. do komunikacji z API i obsługi logiki po stronie klienta

⁵² Atak polegający na wymuszeniu wykonania nieautoryzowanego żądania w imieniu zalogowanego użytkownika. Przeglądarka może automatycznie dołączać dane uwierzytelniające (np. cookies), co umożliwia wykonanie żądania bez wiedzy użytkownika

instancje wyłącznie tego serwisu, nie dotykając pozostałych. To rozwiązanie daje przewagę nad architekturą monolityczną, gdzie skalowanie oznacza powielenie całej aplikacji.

Warunkiem, bez którego skalowanie horyzontalne⁵³ nie ma sensu jest bezstanowość⁵⁴ serwisów. Każdy z serwisów systemu skonfigurowany jest jako bezstanowy, więc nie przechowuje danych sesji użytkownika na swojej stronie. Informacje na temat użytkownika zawarte są w kluczu uwierzytelniającym JWT dołączanym do każdego żądania, dzięki czemu dowolna instancja systemu może obsłużyć dowolne zapytanie bez potrzeby synchronizacji stanu z pozostałymi instancjami.

Asynchroniczne przetwarzanie zdarzeń

Operacje, które nie wymagają natychmiastowego wyniku takie jak generowanie dokumentów PDF, wysyłka powiadomień e-mail, obsługa zdarzeń płatniczych realizowane są asynchronicznie⁵⁵ za pomocą kolejki wiadomości oraz zdarzeń zaimplementowanych wewnątrz serwisów. Producent zdarzenia (np. rental-service po zawarciu umowy) publikuje komunikat do kolejki i natychmiast kontynuuje działanie, nie blokując się na czas przetwarzania. Konsument (np. notification-service) odbiera wiadomości w swoim tempie, niezależnie od producenta. Drugim przykładem jest implementacja reagowania na zdarzenia dziejące się na encjach w bazie takich jak AFTER_COMMIT, gdzie po takiej akcji publikowane jest wewnętrzne zdarzenie, które możemy przejąć konkretną metodą serwisową i obsłużyć dalsze operacje. Taki model pozwala na swobodne skalowanie konsumentów przez dodawanie kolejnych instancji nasłuchujących na tej samej kolejce.

Przetwarzanie partiami i współbieżne operacje

W systemie istnieje kilka miejsc, w których możliwy jest potencjalny duży nakład operacji w jednym momencie, takich jak wysyłka powiadomień, czy obsługa automatycznych zmian statusów. Odpowiedzialnymi mechanizmami za to są moduły planowania, czyli tak zwane schedulery. Obsługują zdarzenia cyklu życia najmu (przypomnienie, zakończenie umów, start najmów) i są przystosowane do przetwarzania wielu rekordów w partiach jednym czasie. Zapobiega to nadmiernemu zużyciu pamięci przy jednoczesnym przetwarzaniu dużej liczby rekordów oraz chroni zewnętrzne serwisy przed nagłym przyrostem zapytań.

Buforowanie danych

W modułach zajmujących się współbieżnymi partiami danych, które w każdym cyklu przetwarzają wiele umów najmu, dane wynajmujących i najemców są buforowane w pamięci operacyjnej przy użyciu biblioteki Caffeine Cache⁵⁶ ze względu na pobieranie danych z innych mikroserwisów. Przy pierwszym żądaniu dane są pobierane z zewnętrznego serwisu i zapisywane w cache'u - każde kolejne żądanie o ten sam rekord jest obsługiwane lokalnie, bez

⁵³ Skalowanie horyzontalne - (z. ang horizontal scaling, scale out) - metoda zwiększania wydajności systemu poprzez dodawanie kolejnych instancji serwisu zamiast zwiększania zasobów pojedynczej maszyny.

⁵⁴ Bezstanowość (z. ang statelessness) - cecha systemu polegające na tym, że serwer nie przechowuje żadnych informacji o stanie klienta między kolejnymi żądaniami. Każde żądanie musi zawierać wszystkie dane niezbędne do jego obsługi.

⁵⁵ Model wykonania operacji, w którym zadania nie są realizowane natychmiast i sekwencyjnie, lecz mogą być wykonywane równolegle lub z opóźnieniem, bez blokowania głównego przepływu programu

⁵⁶ Caffeine - biblioteka cache'owania in-memory dla Javy implementująca algorytm W-TinyLFU

kosztownego wywołania sieciowego. Mechanizm ten istotnie redukuje liczbę żądań kierowanych do authorization-service i tenant-service podczas masowego przetwarzania.

Konteneryzacja jako fundament skalowalności systemu

Wszystkie serwisy posiadają pliki konfiguracyjne, które podczas wdrażania budują dockerowe obrazy odpowiednika takiego poszczególnego serwisu i pozwalają na konteneryzację Docker⁵⁷, co umożliwia ich uruchamianie w dowolnej liczbie instancji na dowolnej infrastrukturze. Konteneryzacja stanowi podstawę dla platform orkiestracji kontenerów takich jak Kubernetes⁵⁸, które automatyzują skalowanie w odpowiedzi na aktualne obciążenie systemu - choć wdrożenie na tego typu platformie nie jest częścią bieżącej implementacji, architektura systemu w pełni je umożliwia.

4.3.3 Wydajność

Buforowanie w pamięci operacyjnej

Najbardziej kosztowną operacją w rozproszonym systemie mikroserwisowym jest wywołanie sieciowe do innego serwisu. Aby ograniczyć ich liczbę, system stosuje buforowanie w pamięci operacyjnej przy użyciu biblioteki Caffeine Cache.

Buforowanie jest przede wszystkim w komponentach przetwarzających duże wolumeny danych np. Schedulerach i listenerach zdarzeń płatniczych. Dane wynajmujących i najemców, które są wielokrotnie odpytywane w trakcie przetwarzania, pobierane są z zewnętrznych serwisów tylko raz, a następnie serwowane z lokalnego cache'u dla kolejnych żądań w ramach tego samego cyklu. Eliminuje to wielokrotne wywołania sieciowe dla tych samych danych.

Eliminacja problemu N+1

Klasycznym problemem wydajnościowym w systemach w modelu domenowym korzystających z wielu serwisów i zbiorów jest wzorzec N+1⁵⁹, czyli dla listy N elementów jest N dodatkowych zapytań o dane powiązane. System świadomie go unika przez stosowane zapytań zbiorczych.

Przykładem jest pobieranie listy umów najmu przez wynajmującego: identyfikatory wszystkich najemców z całej listy zbierane są najpierw do zbioru, a następnie wykonywane jest jedno żądanie do tenant-service zwracające dane wszystkich potrzebnych najemców jednocześnie. Analogicznie zdjęcia lokali są pobierane są jednym zapytaniem po identyfikatorze wynajmującego, a następnie grupowane po stronie aplikacji. Dzięki temu liczba operacji przez sieć jest stała i niezależna od liczby elementów na liście.

Strumieniowanie plików

⁵⁷ Docker - platforma do konteneryzacji aplikacji umożliwiająca pakowanie kodu wraz z zależnościami w izolowane kontenery uruchamiane na dowolnym systemie operacyjnym

⁵⁸ Kubernetes - otwarte źródłowa platforma do orkiestracji kontenerów, automatyzująca wdrażanie, skalowanie i zarządzanie skonteneryzowanymi aplikacjami

⁵⁹ Problem N+1 - antywzorzec wydajnościowy polegający na tym, że zamiast pobrać dane jednym zapytaniem do bazy, aplikacja wykonuje 1 zapytanie główne zwracające N rekordów, a następnie dla każdego z nich wykonuje dodatkowe zapytanie - łącznie N+1 zapytań.

Pobieranie dokumentów PDF realizowane jest przez strumieniowanie przy użyciu `StreamingResponseBody`⁶⁰, który pozwala na przesyłanie danych do klienta fragmentami w miarę ich odczytywania ze źródła, bez uprzedniego wczytywania całego pliku do pamięci serwera. Ma to szczególne znaczenie przy równoczesnym pobieraniu wielu plików przez wiele użytkowników, mniejsze zużycie pamięci operacyjnej serwera i nie rośnie proporcjonalnie do liczby obsługiwanych pobrań.

Przetwarzanie współbieżne

Wysyłka powiadomień e-mail w ramach harmonogramów jest przetwarzane współbieżne przy użyciu `CompletableFuture`⁶¹ i dedykowanego `TaskExecutor`. Każda partia rekordów przetwarzana jest w osobnym wątku, co pozwala na równoległe wykonywanie operacji sieciowych do serwisu pocztowego. Czas przetwarzania całego paczki jest zbliżony do czasu przetwarzania jednego elementu, a nie sumy czasów wszystkich elementów.

Leniwe ładowanie danych

Relacje w niektórych miejscach do operacji autoryzujących między encjami bazodanowymi skonfigurowane są z trybem `FetchType.LAZY`⁶². Powiązane dane są pobierane z bazy dopiero w momencie faktycznego dostępu do nich, nie zaś z góry przy każdym załadowaniu encji głównej. Zapobiega to pobieraniu zbędnych danych w scenariuszach, gdzie powiązane kolekcje nie są potrzebne do obsługi danego żądania.

Harmonogram zadań cyklicznych poza godzinami szczytu

Zadania cykliczne uruchamiane są w godzinach wczesnoporannych o 4:30, 5:00 i 6:00 czasu polskiego. Jest to świadomy wybór, a przetwarzanie odbywa się w czasie najmniejszego obciążenia systemu, gdy potencjalna liczba aktywnych użytkowników jest minimalna. Dzięki temu rozwiązaniu zadanie cykliczne nie blokuje zasobów serwera.

Bezstanowość i pula połączeń

Brak przechowywania stanu sesji po stronie serwera eliminuje konieczność dostępu do współbieżnego magazynu sesji przy każdym żądaniu. Połączenie z bazą danych zarządzane są przez pulę połączeń `HikariCP`⁶³, który jest domyślnym mechanizmem Spring Boot uznawanym za jedną z wydajniejszych implementacji connection pool dla Javy. Ponowne wykorzystanie istniejących połączeń zamiast otwierania nowych dla każdego żądania znacząco redukuje narzut czasowy operacji bazodanowych.

⁶⁰ `StreamingResponseBody` - interfejs Spring Framework umożliwiający asynchroniczne przesyłanie danych do klienta HTTP w trybie strumieniowym, bez konieczności buforowania całej odpowiedzi w pamięci serwera

⁶¹ `CompletableFuture` - klasa wprowadzona w Java 8 (pakiet `java.util.concurrent`), reprezentująca asynchroniczne obliczenie, którego wynik będzie dostępny w przyszłości.

⁶² `FetchType.LAZY` - strategia ładowania danych w JPA/Hibernate, w której powiązane encje nie są pobierane z bazy danych w momencie załadowania encji nadrzędnej, lecz dopiero przy pierwszym odwołaniu do nich w kodzie.

⁶³ `HikariCP` - wysokowydajna biblioteka connection pooling dla Javy, uznawana za najszybszą implementację puli połączeń JDBC

4.3.4 Niezawodność

Transakcyjne publikowanie zdarzeń

Krytycznym problemem w systemach rozproszonych jest sytuacja, w której operacja bazodanowa zakończy się sukcesem, ale wysłanie powiązanego zdarzenia (np. maila aktywacyjnego) się nie powiedzie lub odwrotnie. System rozwiązuje ten problem stosując *@TransactionalEventListener* ze Spring Framework.

Zdarzenia domenowe (np. *RentalCreatedEvent*, *CreatedTenantEvent*) są publikowane wyłącznie po pomyślnym zatwierdzeniu transakcji bazodanowej na zdarzeniu *TransactionPhase.AFTER_COMMIT*. Jeśli transakcja się nie powiedzie i zostanie wycofana - zdarzenie *AFTER_ROLLBACK*, system uruchamia akcję kompensującą - przykładem jest metoda konsumenta w tenant-service, który w przypadku błędu wysyła do authorization-service komunikat o konieczności usunięcia właśnie utworzonego konta użytkownika. Jest to metoda kompensująca.

Gwarancja dostarczania komunikatów

Komunikacja asynchroniczna przez RabbitMQ zapewnia trwałość wiadomości, komunikaty opublikowane do kolejki zostaną dostarczone do konsumenta nawet jeśli w momencie publikacji jest ona niedostępna. Notification-service może być chwilowo wyłączony podczas wdrożenia nowej wersji, a po ponownym uruchomieniu przetworzy wszystkie oczekujące w kolejce wiadomości. Eliminuje to ryzyko utraty powiadomień e-mail przy tymczasowej niedostępności serwisu.

Walidacja danych przed przetwarzaniem zdarzeń

Konsument zdarzeń płatniczych *PaymentAMPQListener* przed przystąpieniem do przetwarzania każdego odebranego zdarzenia weryfikuje kompletność wymaganych danych np. obecność faktury. Brak wymaganego pola skutkuje natychmiastowym odrzuceniem zdarzenia z czytelnym komunikatem błędu, zamiast propagacji wyjątku w dalszej logice biznesowej.

Globalna obsługa błędów

Wszystkie serwisy posiadają globalny handler wyjątków *@RestControllerAdvice*, który przechwytyje nieobsłużone wyjątki tłumaczy je na odpowiednie kody http z ustrukturyzowanymi komunikatami błędów. Zapobiega to ujawnianiu wewnętrznych szczegółów implementacji w odpowiedziach API oraz zapewnia spójny format błędów niezależnie od źródła wyjątku.

Bezstanowość

Brak stanu sesji po stronie serwera sprawia, że każda instancja systemu może obsłużyć każde żądanie. Awaria jednej z instancji nie powoduje utraty sesji użytkownika, kolejne zapytania mogą być skierowane do innych dostępnych instancji bez żadnego wpływu na użytkownika. Jest to podstawowy warunek wysokiej dostępności w modelu architektury rozproszonej.

4.3.5 Integracje zewnętrzne

Rejestr firm GUS

Przy rejestracji firmy, która jest podmiotem gospodarczym, system weryfikuje jej podany identyfikator podatkowy na starcie w zewnętrznym systemie API Głównego Urzędu Statystycznego. Integracja realizowana jest przez dedykowanego klienta w module *GusClient* z obsługą sesji *GusSessionManager*. Podczas zalogowania się do API zewnętrznego systemu wymagane jest uprzednie potwierdzenie swojej tożsamości oraz autoryzacja, jak i utrzymanie aktywnej sesji przez cały czas komunikacji. W przypadku braku firmy o podanym numerze identyfikatora podatkowego, system wyświetla komunikat z błędem *NotFoundCompanyExternalException*.

Serwis powiadomień e-mail (SMTP)

Notification-service wysyła wiadomości e-mail poprzez protokół SMTP⁶⁴ przy użyciu *JavaMailSender*, który jest standardową abstrakcją Spring Framework dla komunikacji pocztowej. Treść wiadomości jest generowana za pomocą szablonów HTML przy użyciu silnika Thymeleaf, który pozwala na budowanie czytelnych, sformatowanych wiadomości z dynamicznie wstawianymi danymi bez mieszania logiki z kodem Java.

⁶⁴ (Simple Mail Transfer Protocol) - standardowy protokół komunikacyjny służący do wysyłania wiadomości e-mail pomiędzy serwerami pocztowymi oraz z klienta pocztowego do serwera poczty

5. Zastosowane technologie, narzędzia i standardy

W niniejszym projekcie wykorzystano szereg technologii, narzędzi programistycznych oraz standardów wspierających proces projektowania, implementacji i testowania aplikacji. Dobór poszczególnych rozwiązań został dokonany z uwzględnieniem wydajności, skalowalności, bezpieczeństwa oraz wygody dalszego rozwoju systemu, aby system charakteryzował się generycznością i jakością rozwiązania gotowego do wdrożenia na maszynie z różnymi systemami operacyjnymi.

5.1. Użyte technologie

W realizacji systemu wykorzystano nowoczesne technologie informatyczne, które umożliwiły sprawne zaprojektowanie, implementację oraz testowanie aplikacji. Dobór narzędzi i środowisk programistycznych był podyktowany wymaganiami funkcjonalnymi systemu, jego skalowalnością oraz łatwością dalszego rozwoju.

W projekcie zastosowano zarówno technologie po stronie serwera, odpowiadające za logikę biznesową i przetwarzanie danych, jak i technologie frontendowe, które umożliwiają intuicyjną i przejrzystą obsługę interfejsu użytkownika. Dodatkowo wykorzystano systemy baz danych do efektywnego przechowywania i zarządzania informacjami oraz narzędzia wspierające kontrolę wersji i organizację pracy zespołowej.

5.1.1 Java

Java 21 została wybrana jako język implementacji większości serwisów backendowych systemu [18]. Jest to język statycznie typowany, działający na maszynie wirtualnej JVM⁶⁵. Wersja Java 21 jako wydanie LTS⁶⁶, gwarantuje długoterminowe wsparcie, stabilność oraz regularne aktualizacje bezpieczeństwa, co ma istotne znaczenie w systemach produkcyjnych. Język ten charakteryzuje się również dojrzałym ekosystemem bibliotek oraz narzędzi, które wspierają szybkie tworzenie i utrzymanie rozbudowanych systemów backendowych. Istotnym czynnikiem wyboru była bardzo dobra znajomość technologii w zespole oraz jej ścisła integracja z frameworkiem Spring Boot, który stanowi podstawę warstwy aplikacyjnej systemu. Spring Boot zapewnia wsparcie dla budowy mikroservisów, automatycznej konfiguracji oraz łatwej integracji z komponentami takimi jak bazy danych, systemy kolejkowe czy narzędzia do monitoringu.

5.1.2 Spring Boot

Spring Boot [19] jest najpopularniejszym frameworkiem do tworzenia mikroservisów w języku Java. Opiera się na modelu IoC⁶⁷ i wstrzykiwaniu zależności dostarczanych przez

⁶⁵ Java Virtual Machine - środowisko uruchomieniowe, które wykonuje skompilowany kod Javy (bytecode). Dzięki JVM programy Java mogą działać na różnych systemach operacyjnych bez konieczności ich modyfikacji

⁶⁶ Long-Term Support - wersja oprogramowania objęta długoterminowym wsparciem. W praktyce takie wydania otrzymują poprawki błędów i aktualizacje bezpieczeństwa przez dłuższy czas niż standardowe wersje

⁶⁷ Inversion of Control - zasada w programowaniu, w której kontrola nad przepływem wykonania programu jest „odwrócona” względem klasycznego podejścia: to framework lub kontener (a nie kod aplikacji) zarządza tworzeniem obiektów i wywoływaniem zależności

kontener Spring. Framework umożliwia tworzenia samodzielnych aplikacji z wbudowanym serwisem HTTP *Tomcat* [20], eliminując potrzebę zewnętrznych serwerów aplikacyjnych.

Dodatkowo Spring Boot oferuje mechanizm automatycznej konfiguracji *auto-configuration* - z którego słynie oraz który znacząco upraszcza proces uruchamiania aplikacji poprzez redukcję ilości konfiguracji wymaganej od programisty. Dzięki pakietom startowym⁶⁸ możliwe jest szybkie dołączanie gotowych zestawów zależności dla konkretnych obszarów funkcjonalnych, takich jak dostęp do bazy danych, bezpieczeństwo czy komunikacja REST.

Framework bardzo dobrze wspiera budowę architektury mikroservisowej, umożliwiając łatwą integrację z systemami komunikacji asynchronicznej (np. kolejkami wiadomości), rejestrowaniem usług oraz narzędziami do monitorowania stanu aplikacji *Spring Actuator* [21]. Dzięki temu możliwe jest tworzenie skalowalnych i odpornych na awarie systemów rozproszonych.

5.1.3 C#

Technologia [22] jest statycznie typowanym, obiektowym językiem programowania rozwijanym przez Microsoft w ramach platformy .NET. Charakteryzuje się nowoczesną składnią, wsparciem dla programowania asynchronicznego (*async/await*⁶⁹) oraz silną integracją z ekosystemem .NET. W projekcie zastosowany do implementacji serwisu płatności oraz serwisu generowania dokumentów.

Istotną zaletą platformy .NET jest bardzo dobre wsparcie dla architektury mikroservisowej oraz komunikacji sieciowej. Framework ASP.NET Core umożliwia tworzenie wysokowydajnych usług HTTP i API REST, zapewniając jednocześnie łatwą konfigurację mechanizmów uwierzytelniania, logowania oraz integracji z bazami danych i zewnętrznymi usługami.

Dodatkową zaletą języka C# jest jego wieloplatformowa formuła oraz możliwość uruchamiania aplikacji zarówno w środowisku Windows, jak i Linux. Ułatwia to wdrażanie aplikacji w środowiskach kontenerowych oraz integrację z nowoczesną infrastrukturą chmurową.

5.1.4 .NET

W systemie został zastosowany .NET [23] w wersji 9 z frameworkiem ASP.NET Core [24] wykorzystywany do zaimplementowania serwisu płatności oraz serwisu do generowania dokumentów z gotowych szablonów HTML. ASP.NET Core oferuje wysokowydajny potok HTTP, natywne wsparcie dla DI⁷⁰ oraz minimalne API. W projekcie zastosowano Entity Framework Core jako ORM⁷¹, bibliotekę PuppeteerSharp do generowania dokumentów PDF z stworzonych szablonów HTML.

⁶⁸ Moduły konfiguracyjne, dzięki którym osoba zajmująca się kodem nie musi martwić się o podstawowe ustawienia innych zależności

⁶⁹ Mechanizm asynchronicznego programowania w języku C#, umożliwiający wykonywanie operacji bez blokowania głównego wątku aplikacji oraz prostsze zarządzanie zadaniami asynchronicznymi

⁷⁰ Dependency Injection - wzorzec projektowy polegający na dostarczaniu (wstrzykiwaniu) zależności do obiektów z zewnątrz, zamiast tworzenia ich wewnątrz klasy

⁷¹ Object-Relational Mapping - technika mapowania obiektów z kodu aplikacji na struktury relacyjnej bazy danych, pozwalająca operować na danych bazodanowych za pomocą obiektów zamiast bezpośrednich zapytań SQL

Platforma .NET 9 stanowi nowoczesne środowisko uruchomieniowe umożliwiające budowę skalowalnych i wydajnych aplikacji backendowych. Dzięki wspólnemu środowisku uruchomieniowemu CLR⁷² zapewnia automatyczne zarządzanie pamięcią oraz optymalizację działania aplikacji w czasie wykonywania. Dodatkowo jest odpowiednie do uruchamiania aplikacji zarówno w środowisku Windows, jak i Linux, co jest szczególnie istotne w kontekście wdrożeń kontenerowych.

5.1.5 PostgreSQL

Technologia bazodanowa, która została użyta jako jedna baza w projekcie w wersji 17. Natomiast każdy serwis działa na tabelach należących do jego odpowiedzialności, każda domena komunikuje się ze sobą przez serwisy nie łamiąc zasad DDD. Baza jest uruchomiona w kontenerze dockerowym. PostgreSQL został wybrany ze względu na łatwość użytkowania oraz znajomość ze studiów i pracy zawodowej.

PostgreSQL [25] jest zaawansowanym, obiektowo-relacyjnym systemem zarządzania bazą danych typu open-source. Charakteryzuje się wysoką zgodnością ze standardem SQL oraz dużą stabilnością i niezawodnością w środowiskach produkcyjnych. System ten wspiera transakcje zgodne z ACID⁷³, co zapewnia spójność i integralność danych nawet w przypadku równoczesnych operacji wykonywanych przez wiele serwisów.

W kontekście architektury mikroservisowej istotną cechą tej bazy danych jest możliwość logicznego rozdzielenia danych pomiędzy serwisy przy zachowaniu jednej fizycznej instancji bazy danych. Ułatwia to zarządzanie infrastrukturą, jednocześnie pozwalając na utrzymanie separacji odpowiedzialności zgodnie z zasadami domeny.

5.1.6 Angular

Angular [26] jest frameworkiem frontendowym firmy Google opartym na Typescript. Przeznaczony jest do budowania aplikacji SPA⁷⁴. W projekcie używany jest do implementacji dwóch projektów: landlord-frontend (panel wynajmującego) i tenant-frontend (panel najemcy).

Ta technologia opiera się obecnie na architekturze komponentowej, w której interfejs użytkownika jest budowany z niezależnych, wielokrotnego użytku komponentów. Dzięki temu możliwe jest łatwiejsze zarządzanie złożonością aplikacji oraz utrzymanie spójności kodu w większych projektach. Budowany system korzysta z Angulara w wersji 19.2, która jest wersją w fazie przejścia z modułów na pojedyncze komponenty standalone⁷⁵.

Framework zapewnia również rozbudowany system wstrzykiwania zależności, routing po stronie klienta oraz wsparcie dla reaktywnego programowania z wykorzystaniem odpowiednich bibliotek. Pozwala to na efektywną obsługę asynchronicznych operacji, takich jak komunikacja z API backendowym.

⁷² Common Language Runtime to podstawowa część platformy .NET, czyli środowisko uruchomieniowe, w którym wykonywane są programy napisane w językach takich jak C#

⁷³ Zestaw czterech właściwości transakcji w bazach danych: Atomicity (atomowość), Consistency (spójność), Isolation (izolacja), Durability (trwałość). Gwarantują one, że transakcje w systemie bazodanowym są przetwarzane niezawodnie, nawet w przypadku błędów

⁷⁴ Single Page Application - architektura aplikacji webowych, w której cała aplikacja działa w ramach jednej strony HTML, a zmiany widoków odbywają się dynamicznie w przeglądarce bez pełnego przeładowania strony

⁷⁵ Podejście w Angularze, w którym komponenty, dyrektywy i serwisy mogą działać bez konieczności deklarowania ich w modułach

5.1.7 Typescript

TypeScript [27] jest nadzbiorem języka JavaScript opracowanym przez Microsoft, wprowadzającym statyczne typowanie oraz zaawansowane mechanizmy obiektywne. Kompiluje się do czystego JavaScriptu, zapewniając jednocześnie wykrywanie błędów na etapie kompilacji oraz lepsze wsparcie narzędzi deweloperskich. Zastosowanie tej technologii w warstwie frontendowej zwiększa czytelność i utrzymanie kodu.

Dzięki systemowi typów TypeScript umożliwia definiowanie interfejsów, typów złożonych oraz generycznych, co pozwala na budowanie bardziej przewidywalnych i skalowalnych aplikacji frontendowych. W dużych projektach znacząco ogranicza to liczbę błędów podczas działania aplikacji oraz ułatwia zmiany kodzie.

Istotną zaletą TypeScriptu jest również jego ścisła integracja z nowoczesnymi frameworkami frontendowymi, takimi jak Angular, React czy Vue, co sprawia, że jest on obecnie jednym z najczęściej wybieranych języków do budowy aplikacji webowych o dużej skali.

5.1.8 RabbitMQ

RabbitMQ [28] jest brokerem wiadomości implementującym protokół AMQP. W systemie pełni rolę szyny zdarzeń domenowych. Producenci zdarzenia publikują je (np. płatność, generowanie dokumentów) do exchange⁷⁶, skąd RabbitMQ przekierowuje dane obiekty zdarzeń do odpowiednich kolejek subskrybentów. Zastosowanie wzorca publish-subscribe zapewnia luźne powiązanie między serwisami.

Technologia ta umożliwia asynchroniczną komunikację pomiędzy komponentami systemu, co znacząco zwiększa jego skalowalność oraz odporność na przeciążenia. Dzięki zastosowaniu komunikacji opartej na wiadomościach serwisy nie muszą oczekiwać na natychmiastową odpowiedź innych usług, co redukuje sprzężenie czasowe między nimi.

5.2. Narzędzia i biblioteki

5.2.1 Leaflet.js

Leaflet.js jest lekką biblioteką JavaScript, która jest przeznaczona do tworzenia interaktywnych map w aplikacjach webowych [29]. Biblioteka oferuje obsługę kafelkowych warstw map, markerów⁷⁷, tooltipów⁷⁸, zdarzeń myszy oraz grupowania obiektów geograficznych. Charakteryzuje się niskim rozmiarem paczki oraz prostym API, co czyni ją popularnym wyborem w zastosowaniach wymagających podstawowej funkcjonalności mapowej bez narzutu rozbudowanych frameworków GIS⁷⁹. W systemie ten mechanizm zastosowany jest w aplikacji klienckiej wynajmującego w dwóch miejscach: na głównym z

⁷⁶ Komponent w systemie kolejkowania wiadomości, który odpowiada za odbieranie wiadomości od producentów i ich routowanie do odpowiednich kolejek na podstawie reguł

⁷⁷ Obiekt biblioteki Leaflet reprezentujący znacznik umieszczany na mapie

⁷⁸ Element interfejsu wyświetlający dodatkową informację po najechaniu kursorem lub kliknięciu na obiekt mapy

⁷⁹ Środowiska programistyczne i systemy wspierające tworzenie, przetwarzanie i analizę danych przestrzennych w ramach systemów informacji geograficznej. Obejmują m.in. biblioteki i platformy takie jak ArcGIS czy QGIS

widokiem wszystkich nieruchomości oraz w formularzu dodawania/edycji nieruchomości podczas wyboru współrzędnych konkretnego mieszkania.

5.2.2 RxJs

RxJS (Reactive Extension for Javascript) [30] jest biblioteką implementującą wzorce reaktywne *ReactiveX* [31] oparty na strumieniach danych *Observable*⁸⁰. Zamiast jednorazowych operacji zwracających wartość, są stosowane strumienie, na których można stosować łańcuchy operatorów transformujących, filtrujących i łączących dane. Do obsługi wykorzystywany jest *Observer*, który definiuje reakcję na emitowane wartości, błędy oraz zakończenie strumienia. Są one używane dla żądań http, zdarzeń użytkownika oraz zmiany stanu. Angular używa RxJS jako fundament dla *HttpClient*, routera i formularzy reaktywnych.

5.2.3 NgRx

NgRx [32] to biblioteka do zarządzania stanem dla Angulara oparta na wzorcu *Redux*. Centralna pamięć przechowuje niemutowany stan aplikacji jako pojedynczy obiekt. Każda zmiana stanu zachodzi wyłącznie przez *dispatch*⁸¹ akcji, gdzie *reducer* na jego podstawie oblicza nowy stan, nigdy nie modyfikując poprzedniego. Takie rozwiązanie powoduje przewidywalność przepływu danych. Logika modyfikacji stanu znajduje się w *reducers*, które na podstawie aktualnego stanu oraz otrzymanej akcji zwracają nową wersję stanu. Dzięki temu stan aplikacji pozostaje niemutowalny.

W przypadku operacji asynchronicznych, takich jak komunikacja z API, wykorzystywane są *Effects*, które pozwalają na obsługę efektów ubocznych poza *reducers*. Uzupełnieniem mechanizmu są selektory, umożliwiające wygodne i wydajne pobieranie danych ze store.

5.2.4 Mockito

Mockito to biblioteka do tworzenia obiektów zastępczych (mocków) [33] w testach jednostkowych Java. Pozwala zastąpić prawdziwe zależności (np. repozytorium, serwisy zewnętrzne) kontrolowanymi obiektami, które zwracają z góry zdefiniowane wartości. Dzięki temu rozwiązaniu, można testować wyłącznie logikę testowanej klasy, bez potrzeby uruchamiania bazy danych, kolejki czy zewnętrznego API. W stworzonym projekcie Mockito jest używane w serwisach: *rental-service*, *authorization-service* i *notification-service*.

Spy - częściowe mockowanie

Metoda *spy()* tworzy obiekt oparty na prawdziwej klasie, ale pozwala nadpisać wybrane metody. Użyto go w teście aktualizacji uprawnień, kiedy encja musi być podczas testów prawdziwym obiektem, co pozwala na możliwe weryfikowanie operacji na takim obiekcie.

W przeciwieństwie do metody *mock()*, która tworzy całkowicie sztuczny obiekt bez prawdziwej logiki, *spy()* opiera się na realnym obiekcie. Oznacza to, że domyślnie wszystkie metody działają tak jak w normalnej klasie, chyba że zostaną jawnie nadpisane w teście. Dzięki

⁸⁰ Mechanizm programowania reaktywnego wykorzystywany w Angular do obsługi asynchronicznych strumieni danych

⁸¹ Metoda używana w bibliotece NgRx do emitowania akcji do store'a. Wywołanie jej informuje system zarządzania stanem, że wystąpiło zdarzenie, które może zostać obsłużone przez konkretne moduły

temu można częściowo korzystać z prawdziwej implementacji, a jednocześnie kontrolować kluczowe fragmenty zachowania.

Ta metoda jest szczególnie przydatna w sytuacjach, gdy testujemy bardziej złożone obiekty i chcemy sprawdzić ich interakcje bez rezygnowania z rzeczywistej logiki klasy. Umożliwia również weryfikowanie wywołania metod oraz ich liczby, co pozwala upewnić się, że dana logika została wykonana zgodnie z oczekiwaniami.

5.2.5 Liquibase

Liquibase jest narzędziem do wersjonowania schematu oraz migracji bazy danych udostępnionym w repozytorium maven do zaimportowania [34]. Zamiast ręcznie wykonywać skrypty SQL, każda zmiana w bazie (nowa tabela, kolumna, indeks) jest zapisana w pliku o powszechnej nazwie changelog⁸². Liquibase śledzi, które zmiany w danym pliku zostały już wykonane i przy każdym starcie aplikacji obsługuje tylko te, które nie zostały wcześniej dodane. Dzięki temu baza jest synchronizowana z kodem oraz tym samym praca w zespole staje się ułatwiona.

W projekcie rental-service Liquibase jest skonfigurowany ze Spring Boot i bazą PostgreSQL i służy jako główny moduł zarządzający całą bazą w systemie.

Konfiguracja

W pliku *application.yml*⁸³, pokazanym na listingu 1 jest wskazany plik, który odpowiada za główny moduł wykonawczy z bazą danych i który liquibase czyta jako poprawną konfigurację zarządzania:

Listing 1: Podstawowa konfiguracja liquibase

```
//Część konfiguracyjna w pliku application.yml odpowiadająca za liquibase w systemie
```

```
spring:
  liquibase:
    change-log: classpath:/db/changelog-master.xml
    enabled: true
  jpa:
    hibernate:
      ddl-auto: validate
```

Plik *changelog-master.xml* przedstawia listę wszystkich obsługiwanych changelogów w kolejności ich wykonania. Każdy changelog dotyczy jednej funkcjonalności, co pokazuje listing 2.

⁸² Plik definiujący historię zmian w bazie danych. Zawiera tzw. changesety, czyli opisowe instrukcje zmian schematu (np. tworzenie tabel, modyfikacje kolumn, dodawanie indeksów), które Liquibase wykonuje w kontrolowany i odwracalny sposób

⁸³ Plik konfiguracyjny aplikacji Spring Boot zapisany w formacie YAML. Służy do definiowania ustawień środowiska

Listing 2: Konfiguracja changelog

```
<databaseChangeLog>
  <!-- ... -->
  <include file="/db/changelogs/changelog-RPP-9-initialize-user-and-authority-
table.xml"/>
  <include file="/db/changelogs/changelog-RPP-14-initialize-landlord-domain-
tables-and-seq.xml"/>
  <include file="/db/changelogs/changelog-RPP-112-build-bank-account-
entity.xml"/>
  <!-- ... -->
</databaseChangeLog>
```

5.2.6 PuppeteerSharp

PuppeteerSharp jest biblioteką .NET, która jest portem Javascript'owego Puppeteer [35]. Używa Headless Chrome'a do renderowania stron HTML jako pliku PDF. Biblioteka pozwala na wykonywanie typowych operacji w przeglądarce, takich jak nawigacja po stronach internetowych, interakcja z elementami DOM, wypełnianie formularzy czy wykonywanie zrzutów ekranu. Dzięki temu możliwe jest odtwarzanie zachowań użytkownika w sposób automatyczny. PuppeteerSharp jest często wykorzystywany w testach automatycznych aplikacji webowych, generowaniu dokumentów PDF ze stron internetowych oraz w procesach web scrapingu i automatyzacji zadań przeglądarkowych. W systemie mikroserwis document-service odpowiedzialny jest do generowania dokumentów PDF np. umów najmów i faktur. Są one tworzone na podstawie szablonów HTML i wypełniane danymi, które są do nich wstrzyknięte.

5.2.7 Entity Framework

Entity Framework Core to ORM dla .NET. Daje możliwość operowania na bazie danych przez obiekty C# zamiast pisania czystego kodu i skryptów SQL [36]. Zapytania są pisane za pomocą LINQ⁸⁴, które jest tłumaczone na SQL automatycznie. W projekcie document-service oraz payment-service *EF Core* służy do odczytu danych z bazy systemu PostgreSQL. Podstawową ideą Entity Framework jest odwzorowanie tabel bazy danych na klasy w kodzie aplikacji, a rekordów na obiekty tych klas. Dzięki temu programista operuje na modelu obiektowym, a nie na niskopoziomowych zapytaniach do bazy danych.

Framework udostępnia mechanizm *DbContext*, który pełni rolę głównego punktu dostępu do danych oraz zarządza połączeniem z bazą i śledzeniem zmian encji. Zmiany w obiektach mogą być automatycznie wykrywane i zapisywane do bazy danych za pomocą metody *SaveChanges()*.

Entity Framework wspiera również zapytania LINQ, co pozwala na tworzenie zapytań w języku C# w sposób czytelny i typowany. Dodatkowo oferuje mechanizmy migracji, które umożliwiają wersjonowanie struktury bazy danych i jej automatyczną aktualizację.

⁸⁴ LINQ (*Language Integrated Query*) - mechanizm wbudowany w platformę .NET umożliwiający formułowanie zapytań do różnych źródeł danych (kolekcje obiektów, bazy danych, XML) bezpośrednio w składni języka C#.

5.2.8 Hibernate

Hibernate [37] to ORM dla Javy, który mapuje klasy na tabele bazy danych i pozwala wykonywać na danych operacje zamiast pisać czysty skrypt SQL. W projekcie jest używany z pomocą implementacji JPA w serwisach rental-service, authorization-service i tenant-service. Podstawową ideą Hibernate jest umożliwienie pracy na obiektach języka Java zamiast bezpośredniego operowania na zapytaniach SQL. Klasy aplikacji są odwzorowywane na tabele bazy danych, a ich instancje odpowiadają rekordom w tych tabelach.

Framework udostępnia mechanizm *Session*, który odpowiada za komunikację z bazą danych oraz zarządzanie cyklem życia encji. Zmiany w obiektach są automatycznie wykrywane i mogą być zapisywane w bazie danych poprzez operację zatwierdzenia transakcji. Istotną zaletą Hibernate jest zarządzanie kontekstem trwałości (Persistence Context), które pozwala śledzić zmiany w obiektach znajdujących się w pamięci aplikacji. W momencie zatwierdzenia transakcji framework automatycznie synchronizuje stan obiektów z bazą danych

Hibernate umożliwia również tworzenie zapytań w języku HQL (Hibernate Query Language), który jest niezależny od konkretnego dialektu. Dzięki mechanizmom cache'owania, zarządzania relacjami oraz automatycznego mapowania, Hibernate znacząco upraszcza warstwę dostępu do danych w aplikacjach Java i jest jednym z najczęściej stosowanych ORM w tym ekosystemie SQL, co zwiększa przenośność aplikacji między różnymi systemami bazodanowymi.

5.2.9 Docker

Docker jest narzędziem służącym do konteneryzacji aplikacji [38]. Kontener stanowi izolowane środowisko uruchomieniowe, które zawiera aplikację wraz ze wszystkimi jej zależnościami, bibliotekami oraz konfiguracją potrzebną do działania. Dzięki temu aplikacja może być uruchamiana w identyczny sposób niezależnie od środowiska hosta i systemu operacyjnego, na którym działa Docker.

Podstawowym elementem pracy z Dockerem jest obraz kontenera, który nazwany jest jako (image), czyli zapis gotowego środowiska aplikacji. Na podstawie obrazu tworzone są następnie uruchamialne kontenery. Proces budowy obrazu definiowany jest w pliku *Dockerfile*, który zawiera instrukcje opisujące m.in. wykorzystywany system bazowy, kopiowanie plików aplikacji, instalację zależności oraz sposób uruchamiania programu.

W przypadku aplikacji składających się z wielu usług, takich jak baza danych, backend czy frontend, wykorzystywany jest plik docker-compose.yml. Umożliwia on zdefiniowanie oraz jednoczesne uruchamianie wielu współpracujących ze sobą kontenerów. Dzięki temu możliwe jest łatwe zarządzanie całą infrastrukturą aplikacji oraz automatyzacja procesu jej uruchamiania i konfiguracji.

Multi-stage build - Java (Spring Boot)

Obraz projektu w naszym ekosystemie budowane są dwuetapowo tak jak pokazane jest to na listingu 3.

Listing 3: Konfiguracja budowania obrazu w rejestrze

```

FROM maven:3.9-eclipse-temurin-21 AS builder
WORKDIR /build

COPY pom.xml .
RUN --mount=type=secret,id=maven_settings,target=/root/.m2/settings.xml \
    mvn -s /root/.m2/settings.xml -B -DskipTests dependency:go-offline

COPY src/ ./src
RUN --mount=type=secret,id=maven_settings,target=/root/.m2/settings.xml \
    mvn -s /root/.m2/settings.xml -B -DskipTests package

FROM eclipse-temurin:21-jre
RUN useradd -r -u 1001 -g root appuser && mkdir -p /opt/build && chown -R
appuser:root /opt/build
USER appuser
WORKDIR /opt/build

COPY --from=builder /build/target/rental-service-0.0.1-SNAPSHOT.jar
/opt/build/app.jar

ENV JAVA_OPTS="-XX:+UseContainerSupport -XX:MaxRAMPercentage=75.0 -
XX:InitialRAMPercentage=50.0 -XX:+ExitOnOutOfMemoryError"
EXPOSE 8082
ENTRYPOINT ["sh", "-c", "java $JAVA_OPTS -jar /opt/build/app.jar"]

```

Docker Compose - uruchamianie serwisu

Każdy mikroservis ma własny plik docker-compose.yml podłączający go do wspólnej sieci komunikacyjnej *rental_platform_network* w systemie. Konfiguracja środowiskowa (hosty, URL-e innych serwisów) podawana zostaje przez zmienne środowiskowe, gdzie każdy obraz jest umieszczony pojedynczo w swoim dedykowanym kontenerze. Po zbudowaniu takiego kontenera, uruchomienie konkretnego serwisu staje się bardzo proste.

Listing 4: Konfiguracja w pliku docker-compose.yml serwisu oraz jego zmiennych środowiskowych

```

services:
  rental-service:
    image: registry.gitlab.com/rental-platform-project/rental-service:latest
    container_name: rental-service
    environment:
      DATASOURCE_HOST: postgres:5432
      SPRING_RABBITMQ_HOST: rabbitmq
      SERVICES_TENANT_SERVICE_URL: http://tenant-service:8087
    networks:

```

```

    - rental_platform_network
ports:
    - "8082:8082"

networks:
    rental_platform_network:
        external: true

```

5.2.10 Kubernetes

Kubernetes (k8s) [39] to system do orkiestracji kontenerów, który automatycznie uruchamia, restartuje i skaluje aplikacje działające w Dockerze. Każdy zasób jest opisany plikiem konfiguracyjnym YAML⁸⁵. W projekcie została użyta wersja o nazwie k3s, która jest lekką dystrybucją Kubernetesa przeznaczoną do środowisk o ograniczonych zasobach. Każdy mikroservis ma własny zestaw manifestów⁸⁶.

Podstawową jednostką uruchomieniową w Kubernetesie jest, pod⁸⁷ który może zawierać jeden lub kilka współdzielących zasoby kontenerów. Kubernetes zarządza również komunikacją pomiędzy usługami, równoważeniem obciążenia ruchu sieciowego oraz automatycznym odtwarzaniem aplikacji w przypadku awarii.

Konfiguracja zasobów w Kubernetesie opisywana jest za pomocą plików YAML, nazywanych manifestami. Manifesty definiują m.in. sposób uruchamiania kontenerów, liczbę replik aplikacji, konfigurację sieci, zmienne środowiskowe oraz polityki skalowania.

Deployment

Deployment w Kubernetes jest zasobem odpowiedzialnym za deklaracyjne zarządzanie cyklem życia aplikacji. Definiuje on pożądaną liczbę replik⁸⁸ podów oraz sposób ich uruchamiania, aktualizacji i utrzymania. Dzięki Deploymentowi Kubernetes automatycznie monitoruje stan aplikacji i w razie awarii jednego z podów uruchamia jego nową instancję, aby utrzymać zadaną dostępność systemu. Deployment umożliwia również bezpieczne aktualizacje aplikacji, czyli stopniowe zastępowanie starych wersji nowymi bez przerywania działania usługi, a także łatwe przywracanie w przypadku błędów wdrożenia.

Service

Service to zasób Kubernetes, który zapewnia stabilny punkt dostępu do aplikacji działających w klastrze. Ponieważ pody są dynamiczne i mogą być tworzone oraz usuwane, ich adresy IP⁸⁹

⁸⁵ Tekstowy format serializacji danych oparty na czytelnej dla człowieka składni wcięć. Powszechnie stosowany w plikach konfiguracyjnych aplikacji oraz narzędzi DevOps (Docker Compose, Kubernetes, GitHub Actions) jako alternatywa dla XML i JSON

⁸⁶ Plik konfiguracyjny w formacie YAML używany w Kubernetes do deklaratywnego opisu zasobów klastra

⁸⁷ Podstawowa jednostka uruchomieniowa w Kubernetes, stanowiąca grupę jednego lub kilku kontenerów, które współdzielą te same zasoby sieciowe oraz pamięć masową

⁸⁸ Pojedyncza instancja podów uruchomiona w klastrze Kubernetes, będąca kopią tej samej aplikacji. Repliki są wykorzystywane do zwiększania dostępności i odporności systemu na awarie

⁸⁹ Internet Protocol address - unikalny adres sieciowy przypisywany urządzeniom lub usługom w sieci komputerowej, umożliwiający ich identyfikację oraz komunikację

również ulegają zmianie - Service rozwiązuje ten problem, udostępniając stały adres DNS oraz IP, który kieruje ruch do aktualnie działających podów. Service pełni także funkcję load balancera, rozdzielając ruch sieciowy pomiędzy dostępne instancje aplikacji. W zależności od typu, może on udostępniać aplikację wewnątrz klastra, na zewnątrz lub przez zewnętrzny adres IP.

5.2.11 Traefik

Traefik jest działającym mechanizmem reverse proxy⁹⁰ oraz load balance’rem działającym jako Ingress Controller [40] w klastrze Kubernetes. Jego zadaniem jest przyjmowanie ruchu http z zewnątrz klastra i kierowanie go do odpowiednich serwisów na podstawie ścieżek URL. W projekcie Traefik jest domyślnym kontrolerem k3s i obsługuje routing do wszystkich frontendów i backendów aplikacji.

Ingress

Ingress to mechanizm Kubernetesa definiujący reguły routingu. Adnotacja `kubernetes.io/ingress.class: traefik` wskazuje, że ma go obsłużyć Traefik. Implementację mechanizmu pokazuje listing 5. Mechanizm ten pozwala odpowiednio udostępnić aplikacje klienckie na odpowiednich portach, za pomocą których będzie wchodził ruch z zewnątrz do warstwy serwerowej.

Listing 5: Konfiguracja ingress dla aplikacji klienckiej

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: landlord-frontend-ingress
  annotations:
    kubernetes.io/ingress.class: traefik
spec:
  rules:
    - http:
        paths:
          - path: /landlord
            pathType: Prefix
            backend:
              service:
                name: landlord-frontend-service
                port:
                  number: 80
```

5.2.12 Gitlab CI/CD

⁹⁰ Reverse proxy - serwer pośredniczący przyjmujący zapytania od klientów i przekazujący je do odpowiednich serwerów backendowych, ukrywając ich szczegóły infrastrukturalne. Umożliwia realizację load balancingu, terminacji SSL oraz cachowania odpowiedzi.

Jest to system automatyzacji procesów wytwarzania i wdrażania oprogramowania, wbudowany bezpośrednio w platformę *GitLab*. Mechanizm ten umożliwia automatyczne wykonywanie zdefiniowanych zadań po każdej zmianie w repozytorium, co znacząco usprawnia proces dostarczania aplikacji i ogranicza ryzyko błędów wynikających z ręcznego wykonywania kolejnych etapów wdrożenia. Dzięki integracji z systemem kontroli wersji możliwe jest powiązanie procesu budowania i publikacji aplikacji z konkretnymi zmianami w kodzie źródłowym.

Podstawowym elementem **GitLab CI/CD** jest pipeline, uruchamiany automatycznie po wykonaniu operacji `git push` lub po utworzeniu `merge request`. Pipeline stanowi uporządkowaną sekwencję zadań (`jobs`), wykonywanych zgodnie z konfiguracją zapisaną w pliku `.gitlab-ci.yml`, znajdującym się w katalogu głównym projektu. Plik ten definiuje poszczególne etapy procesu, zależności między zadaniami, warunki ich uruchamiania, a także środowisko wykonawcze, w którym uruchamiane są komendy.

Listing 6: Konfiguracja Gitlab CI/CD przedstawiająca kolejne etapy automatyzacji

```
image: maven:3.9.6-eclipse-temurin-21

stages:
  - test
  - build
  - deploy

variables:
  IMAGE_LATEST: $CI_REGISTRY_IMAGE:latest

cache:
  paths:
    - .m2/repository

workflow:
  rules:
    - if: '$CI_COMMIT_BRANCH == "master" && $CI_PIPELINE_SOURCE == "push"'
      when: always
    - when: never
```

Etapy pipeline zaimplementowane w projektach

- Pierwszy etap, **test**, odpowiada za automatyczne uruchomienie testów aplikacji po każdej zmianie w repozytorium. Jego zadaniem jest weryfikacja poprawności kodu oraz szybkie wykrywanie błędów przed przejściem do kolejnych etapów pipeline'u,
- Drugi etap, **build**, obejmuje kompilację projektu z wykorzystaniem narzędzia Maven oraz przygotowanie artefaktu aplikacji. W tym kroku tworzony jest również obraz kontenera Docker, który następnie oznaczany jest odpowiednim tagiem - w projekcie wykorzystano zmienną `IMAGE_LATEST`, wskazującą na najnowszą wersję obrazu w rejestrze GitLab Container Registry,

- Ostatni etap, **deploy**, odpowiada za automatyczne wdrożenie aplikacji na środowisko docelowe. Na tym etapie pobierany jest aktualny obraz kontenera, a następnie wdrażana jest nowa wersja aplikacji w klastrze k3s. Dzięki temu cały proces – od testów, przez budowanie, aż po publikację – przebiega automatycznie i w sposób powtarzalny.

5.2.13 Docker DinD

Docker DinD (Docker in Docker) jest techniką pozwalającą na uruchomienie demona Dockera wewnątrz kontenera Docker. Rozwiązanie to znajduje zastosowanie przede wszystkim w systemach *CI/CD*, w których proces budowania i publikacji obrazów kontenerów realizowany jest przez runner uruchomiony w środowisku kontenerowym. W takiej konfiguracji kontener wykonujący zadania pipeline'u nie posiada bezpośredniego dostępu do demona Docker działającego na hoście.

W projekcie technika Docker DinD została wykorzystana w ramach *GitLab CI/CD* do automatycznego budowania obrazów kontenerów aplikacji. Podczas wykonywania etapu build *GitLab Runner* uruchamia zadanie wewnątrz kontenera, a równocześnie tworzy dodatkowy kontener pełniący rolę serwisu z aktywnym demonem Dockera. Dzięki temu możliwe jest wykonywanie poleceń takich jak `docker build`, `docker tag` oraz `docker push` bez konieczności udostępniania gniazda Dockera hosta. Takie podejście zwiększa izolację środowiska oraz pozwala zachować pełną niezależność procesu budowania od konfiguracji serwera.

Konfiguracja Docker DinD wymaga kilku podstawowych elementów. Pierwszym z nich jest obraz wykorzystywany przez job pipeline'u - w projekcie zastosowano obraz `docker:24.0`, zawierający klienta Docker CLI. Drugim elementem jest dodatkowy serwis `docker:24.0-dind`, uruchamiany równolegle jako osobny kontener z aktywnym demonem Docker. Trzecim krokiem jest ustawienie zmiennej środowiskowej `DOCKER_HOST`, która wskazuje adres uruchomionego serwisu i umożliwia klientowi Docker komunikację z demonem. W praktyce powoduje to, że polecenia wykonywane w zadaniu CI kierowane są do kontenera DinD zamiast do lokalnego środowiska wykonawczego. Część konfiguracji została pokazana na listingu 7.

Listing 7: Konfiguracja Docker DinD dla projektu

```
build:
  stage: build
  image: docker:24.0
  services:
    - docker:24.0-dind
  variables:
    DOCKER_TLS_CERTDIR: ""
    DOCKER_HOST: tcp://docker:2375
```

5.2.14 Caffeine

Caffeine jest biblioteką przeznaczoną do buforowania danych w pamięci operacyjnej (in-memory cache) [41] w aplikacjach napisanych w języku Java. Mechanizm działania polega na przechowywaniu najczęściej wykorzystywanych danych bezpośrednio w pamięci procesu aplikacji, co eliminuje konieczność każdorazowego odwoływania się do wolniejszych źródeł

danych, takich jak baza danych czy zewnętrzne API. Dzięki temu możliwe jest znaczące zwiększenie wydajności oraz redukcja obciążenia warstwy odczytywania danych.

Biblioteka Caffeine umożliwia precyzyjną konfigurację sposobu zarządzania pamięcią podręczną. Pozwala między innymi na definiowanie limitu rozmiaru cache, strategii usuwania najstarszych lub najrzadziej używanych wpisów LRU⁹¹, a także polityki wygasania danych na podstawie czasu od ostatniego dostępu lub czasu zapisu co dobrze odwzorowane jest w przykładzie [42]. Dodatkowo oferuje mechanizmy automatycznego odświeżania danych oraz asynchronicznego ładowania wartości, co pozwala jeszcze lepiej kontrolować sposób aktualizacji przechowywanych informacji.

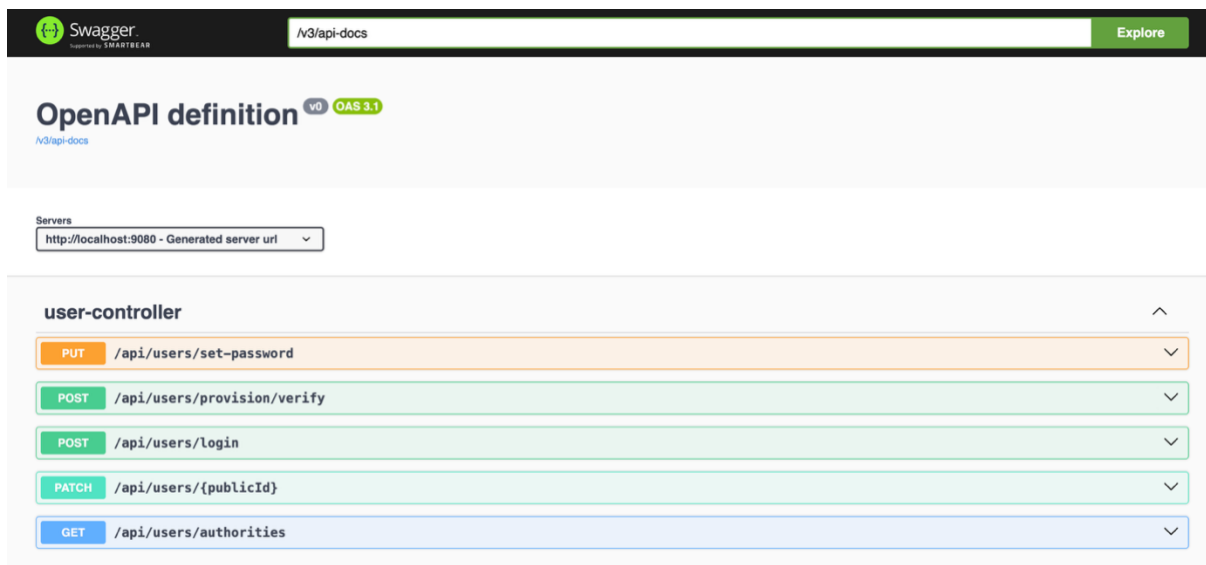
5.2.15 Swagger

Swagger jest zestawem narzędzi opartym na specyfikacji *OpenAPI* [43], służącym do automatycznego opisywania, dokumentowania oraz testowania interfejsów REST API. Jego główną zaletą jest możliwość generowania interaktywnej dokumentacji na podstawie kodu aplikacji, bez konieczności ręcznego tworzenia opisów endpointów. Dzięki temu dokumentacja jest zawsze spójna z aktualną implementacją systemu i aktualizuje się wraz ze zmianami w kodzie źródłowym.

Swagger umożliwia opisanie API w sposób standardowy i czytelny, obejmujący m.in. dostępne endpointy, metody HTTP, parametry wejściowe, struktury odpowiedzi oraz kody statusu. Na podstawie tych definicji generowany jest interaktywny interfejs użytkownika, który pozwala nie tylko przeglądać dokumentację, ale również bezpośrednio wykonywać zapytania testowe do API. Dzięki temu Swagger pełni jednocześnie rolę dokumentacji technicznej oraz narzędzia wspierającego testowanie i integrację.

W projekcie Swagger został wykorzystany jedynie w serwisach pośredniczących z aplikacjami klienckimi udostępniającymi REST API, co znacząco ułatwiło proces testowania oraz integracji. Interaktywna dokumentacja pozwalała na szybkie sprawdzanie działania endpointów, weryfikację struktur danych oraz debugowanie komunikacji pomiędzy usługami. Dzięki temu zmniejszono potrzebę korzystania z zewnętrznych narzędzi testowych oraz przyspieszono proces weryfikacji nowych funkcjonalności.

⁹¹ Least Recently Used - strategia zastępowania danych w pamięci podręcznej (cache), w której usuwany jest element używany najdawniej



Rysunek 4: Strona główna Swagger z opisem endpointów mikroserwisu, Źródło: opracowanie własne

5.2.16 Lombok

Lombok jest biblioteką dla języka Java, której celem jest redukcja tzw. *boilerplate code*, czyli powtarzalnego i schematycznego kodu, który standardowo musiałby być implementowany ręcznie [44]. Biblioteka działa na etapie kompilacji, generując automatycznie dodatkowy kod na podstawie zastosowanych adnotacji w klasach. Obejmuje to między innymi getters, setters, konstruktory, metody *toString()*, *equals()* oraz *hashCode()*.

Zastosowanie Lomboka pozwala znacząco uprościć definicje klas modelowych (np. encji, DTO czy obiektów domenowych). W przypadku klas posiadających wiele pól eliminuje konieczność ręcznego implementowania dużej liczby metod pomocniczych, co nie tylko skraca kod źródłowy, ale również poprawia jego czytelność i utrzymanie. Dzięki temu programista może skupić się na logice biznesowej zamiast na powtarzalnych elementach strukturalnych.

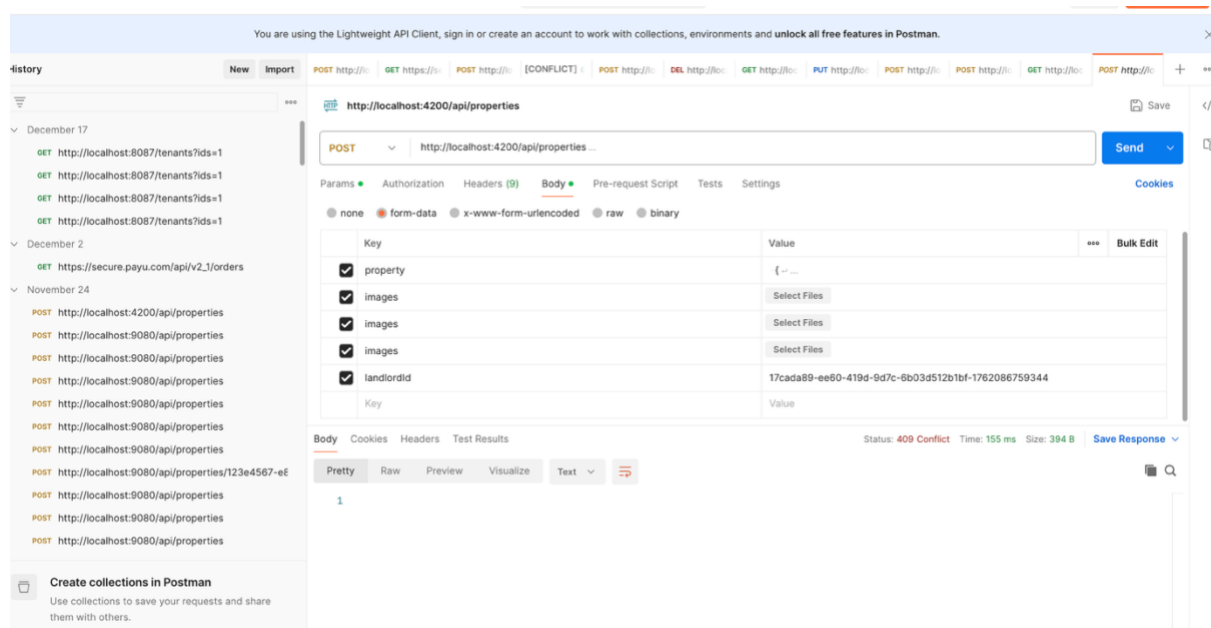
Lombok integruje się z procesem kompilacji projektu, wykorzystując mechanizmy przetwarzania adnotacji. Oznacza to, że wygenerowany kod nie jest widoczny w źródłach, ale jest dostępny w skompilowanej postaci, co nie wpływa negatywnie na działanie aplikacji ani jej wydajność. Przykład użycia tego narzędzie przedstawia listing 40.

5.2.17 Postman

Postman jest narzędziem wykorzystywanym do testowania oraz debugowania interfejsów *REST API*. Umożliwia wysyłanie zapytań HTTP, takich jak *GET*, *POST*, *PUT*, *PATCH* oraz *DELETE*, do dowolnie wybranych endpointów aplikacji. Dzięki graficznemu interfejsowi użytkownika pozwala na łatwe definiowanie szczegółów żądania, w tym nagłówków, parametrów zapytania oraz treści żądania, bez konieczności pisania dodatkowego kodu.

Narzędzie to jest szczególnie przydatne na etapie implementacji oraz testowania usług backendowych, ponieważ umożliwia szybkie sprawdzenie poprawności działania API oraz weryfikację zwracanych danych. Odpowiedzi serwera prezentowane są w przejrzystej formie,

najczęściej jako struktura JSON, wraz z informacjami o kodzie statusu HTTP oraz czasie odpowiedzi. Pozwala to na łatwą analizę działania endpointów oraz identyfikację ewentualnych błędów.



Rysunek 5: Użycie Postmana do testowania endpointów, Źródło: Opracowanie własne

5.2.18 Thymeleaf

Thymeleaf jest silnikiem szablonów dla Javy, który pozwala generować dynamiczne dokumenty HTML po stronie serwera. W projekcie notification-service został użyty do budowania treści wiadomości e-mail, gdzie każdy typ powiadomienia ma własny szablon HTML z dyrektywami, które są wypełniane danymi w czasie rzeczywistym. Główną cechą Thymeleaf jest możliwość pracy zarówno w trybie serwerowym, jak i podglądu statycznego. Oznacza to, że szablony HTML mogą być poprawnie wyświetlane również bez uruchomionej aplikacji, co ułatwia pracę projektową i integrację z warstwą frontendową.

Thymeleaf integruje się z frameworkami takimi jak Spring, umożliwiając wstrzykiwanie danych z warstwy backendowej bezpośrednio do widoków. Dane przekazywane z kontrolerów mogą być renderowane w HTML przy użyciu wyrażeń szablonych, co pozwala na dynamiczne generowanie treści stron.

Biblioteka wspiera m.in. iteracje, warunki logiczne oraz formatowanie danych bez konieczności stosowania dodatkowych skryptów po stronie klienta. Dzięki temu Thymeleaf jest często wykorzystywany w aplikacjach typu MVC jako warstwa prezentacji.

5.3. Zastosowane standardy

W projektowanym systemie wykorzystano powszechnie przyjęte standardy i protokoły komunikacyjne, które stanowią podstawę wymiany danych w nowoczesnych systemach

informatycznych. Ich zastosowanie umożliwia zapewnienie spójnej i niezawodnej komunikacji pomiędzy komponentami systemu oraz ułatwia integrację z zewnętrznymi usługami.

W szczególności wykorzystano protokół HTTP do komunikacji synchronicznej pomiędzy mikroservisami wewnątrz całego ekosystemu oraz SOAP, które zapewniają odpowiednio lekki oraz sformalizowany sposób wymiany informacji. W dalszej części podrozdziału przedstawiono ich charakterystykę oraz zastosowanie w systemie.

5.3.1 HTTP

Komunikacja synchroniczna pomiędzy serwisami oraz między aplikacjami klienckimi, a warstwą serwerową oparta jest na dobrze znanym protokole HTTP, który jest zgodny z architekturą REST. Każdy zasób jest dostępny przez adres URL, a semantyka przedstawiana jest dzięki metodom HTTP: GET do odczytu, PUT do aktualizacji, DELETE, PATCH do częściowej modyfikacji.

W podejściu REST kluczowe znaczenie ma bezstanowość komunikacji (stateless), co oznacza, że każde żądanie HTTP zawiera wszystkie informacje niezbędne do jego obsługi, bez konieczności utrzymywania sesji po stronie serwera. Takie podejście zwiększa skalowalność systemu oraz upraszcza jego architekturę, szczególnie w środowiskach rozproszonych. Dane wymieniane pomiędzy klientem a serwerem są najczęściej przesyłane w formacie JSON, który charakteryzuje się lekką strukturą oraz czytelnością zarówno dla człowieka, jak i maszyn. Dzięki temu możliwe jest łatwe mapowanie struktur danych na obiekty w aplikacjach oraz serwerze.

5.3.2 SOAP

SOAP jest protokołem wymiany komunikatów opartych na formacie XML, stosowanym w architekturze usług sieciowych (Web Services). W odróżnieniu od REST, który opiera się na zasobach i metodach HTTP, SOAP definiuje ścisły kontrakt komunikacyjny przez dokument WSDL⁹². Każdy komunikat SOAP składa się z koperty (envelope) zawierającej nagłówek i ciało. Protokół gwarantuje niezależność od warstwy transportowej, może być przesyłany przez HTTP, SMTP lub inne protokoły. W stworzonym systemie SOAP jest zastosowany wyłącznie do integracji z zewnętrznym rejestrem firm Głównego Urzędu Statystycznego, który udostępnia API tylko w tej technologii.

Klasy zapytań i odpowiedzi (*Zaloguj*, *ZalogujResponse*, *DaneSzukajPodmioty* oraz inne) generowane są automatycznie z pliku WSDL przez narzędzia wsiport i serializowane przez JAXB (Jakarta XML Binding). Klient rozszerza *WebServiceGatewaySupport* dostarczany przez Spring Web Services, który zarządza cyklem życia szablonu SOAP. Protokół ten został zaprojektowany jako uniwersalny standard komunikacji pomiędzy systemami rozproszonymi. Jego głównym założeniem jest zapewnienie ustrukturyzowanego i jednoznacznego sposobu wymiany informacji pomiędzy aplikacjami niezależnie od wykorzystywanego języka programowania czy platformy systemowej. Dzięki zastosowaniu XML komunikaty posiadają zdefiniowaną strukturę oraz mogą być walidowane względem określonych schematów.

Istotną cechą protokołu SOAP jest możliwość rozszerzania komunikatów o dodatkowe informacje umieszczane w nagłówku, takie jak dane uwierzytelniające, podpisy cyfrowe,

⁹² Web Services Description Language - język opisu usług sieciowych oparty na XML, używany do definiowania interfejsu usługi SOAP: dostępnych operacji, parametrów wejściowych i wyjściowych oraz sposobu komunikacji

informacje o transakcjach czy mechanizmy bezpieczeństwa. Umożliwia to realizację bardziej złożonych scenariuszy integracyjnych, szczególnie w systemach biznesowych oraz administracyjnych, w których wymagana jest wysoka niezawodność i formalizacja komunikacji.

5.3.3 Architektura sterowana zdarzeniami (Event Driven Architecture)

Komunikacja asynchroniczna pomiędzy mikroserwisami opiera się na wzorcu **Event-Driven Architecture (EDA)** z wykorzystaniem brokera wiadomości, np. RabbitMQ. Serwisy publikują zdarzenia do exchange⁹³, a nie bezpośrednio do kolejek. Dopiero mechanizmy routingu decydują, które kolejki otrzymają dane zdarzenie.

Producent zdarzeń nie zna konsumentów - jego odpowiedzialnością jest jedynie publikacja komunikatu do brokera. Konsumenti subskrybują interesujące ich zdarzenia niezależnie, co zapewnia luźne powiązanie systemu, czyli loose coupling⁹⁴. Zdarzenie stanowi niezmienny komunikat informujący o tym, że w systemie zaszło określone zdarzenie biznesowe, takie jak utworzenie dokumentu, dokonanie płatności czy rejestracja użytkownika.

W architekturze EDA wyróżnia się trzy podstawowe elementy: producentów zdarzeń, broker komunikatów oraz konsumentów zdarzeń. Producent odpowiada za publikację zdarzeń do systemu komunikacyjnego, nie posiadając wiedzy o liczbie ani typach odbiorców. Broker komunikatów (np. RabbitMQ, Apache Kafka) pełni rolę pośrednika odpowiedzialnego za dystrybucję zdarzeń do odpowiednich odbiorców zgodnie z konfiguracją routingu. Konsumenti subskrybują wybrane typy zdarzeń i reagują na nie w sposób niezależny, realizując własną logikę biznesową

Natomiast wewnątrz pojedynczego serwisu do obsługi zdarzeń domenowych można wykorzystać mechanizm Spring *ApplicationEventPublisher* wraz z adnotacją *@TransactionalEventListener*, najczęściej z fazą AFTER_COMMIT. Dzięki temu zdarzenia są publikowane dopiero po pomyślnym zakończeniu transakcji bazodanowej, co zmniejsza ryzyko niespójności danych.

Architektura sterowana zdarzeniami stanowi podejście szczególnie przydatne w systemach rozproszonych i mikroserwisowych, gdzie istotne są skalowalność, elastyczność oraz niezależność komponentów, kosztem zwiększonej złożoności systemu oraz trudniejszego zarządzania.

⁹³ Komponent pośredniczący w RabbitMQ odpowiedzialny za odbieranie wiadomości od producentów i kierowanie ich do odpowiednich kolejek na podstawie zdefiniowanych reguł routowania

⁹⁴ Zasada w architekturze oprogramowania polegająca na minimalizowaniu zależności między komponentami systemu. Moduły komunikują się ze sobą poprzez jasno zdefiniowane interfejsy, a nie bezpośrednie zależności implementacyjne

6. Projekt i implementacja

Następnym etapem budowania rozwiązania jest faza implementacyjna. W tym momencie zaczynamy skupiać się już na samym produkcie. Opracowane wcześniej wymagania funkcjonalne i niefunkcjonalne, modele procesów biznesowych oraz założenia architektoniczne zostają odwzorowane w postaci rzeczywistych komponentów aplikacji. Implementacja obejmuje zarówno warstwę logiki biznesowej, interfejs użytkownika, jak i mechanizmy komunikacji z bazą danych oraz innymi elementami ekosystemu.

W trakcie realizacji tego etapu szczególną uwagę poświęca się jakości kodu źródłowego, zachowaniu spójności architektonicznej oraz zapewnieniu odpowiedniej wydajności aplikacji. Istotnym aspektem jest również optymalizacja procesów wykonywanych przez system, tak aby zagwarantować stabilność działania oraz możliwość dalszego rozwoju rozwiązania. Wdrażane mechanizmy powinny zapewniać trwałość danych, ich integralność oraz bezpieczny i wydajny dostęp pomiędzy wszystkimi modułami systemu.

Proces implementacji uwzględnia także stosowanie dobrych praktyk programistycznych, takich jak modularność kodu, separacja odpowiedzialności poszczególnych komponentów czy wykorzystanie odpowiednich wzorców projektowych. Dzięki temu rozwijane rozwiązanie staje się bardziej skalowalne, łatwiejsze w utrzymaniu oraz odporne na błędy wynikające z dalszej rozbudowy systemu.

6.1. Architektura mikroserwisowa

W naszym rozwiązaniu systemowym, został przyjęty wzorzec mikroserwisowy, gdzie charakterystycznym elementem, jak sama nazwa wskazuje są małe, odrębne i niezależne serwisy, które razem w połączeniu tworzą cały ekosystem. Oprócz samego podziału, każdy z serwisów jest odpowiedzialny za grupę procesów które są do niego przypisane i które tylko on realizuje. Ten podział odpowiedzialności jest ustrukturyzowany za pomocą podejścia domenowego, które zostało przyjęte na początku badań nad funkcjonalnościami jakie zostaną zaimplementowane do systemu. Podzielenie systemu na odrębne domeny pozwoliło osiągnąć wiele korzyści, ale również też utrudnień i złożoności, o której musimy pamiętać. Korzyścią takiego zastosowania jest oczywiście rozłożenie obciążenia obliczeniowego spowodowanego zapytaniami do serwera, gdzie zapytania są delegowane do serwisu odpowiedzialnego za daną część obliczeń, co pozwala na zwiększenie wydajności, poprzez odciążenie głównego serwisu. Innymi zaletami jakie skutkują po zastosowaniu takiego podziału są:

- **Niezależne skalowanie usług** - jest to możliwość niezależnego skalowania poszczególnych komponentów systemu. W praktyce oznacza to, że usługi obsługujące większą liczbę żądań mogą być skalowane niezależnie od pozostałych elementów aplikacji. Pozwala to efektywniej wykorzystywać zasoby infrastrukturalne oraz ograniczać koszty utrzymania środowiska,
- **Niezależność technologiczna** - umożliwia stosowanie różnych technologii w zależności od wymagań konkretnego komponentu. W ramach projektu wykorzystano zarówno technologię Java ze Spring Boot, jak i platformę .NET. Pozwoliło to dobrać odpowiednie narzędzia do realizacji konkretnych funkcjonalności systemu,

- **Izolacja awarii** - w przypadku wystąpienia błędu w jednym z mikroservisów pozostałe usługi mogą nadal funkcjonować poprawnie. Ogranicza to ryzyko całkowitego zatrzymania systemu oraz zwiększa jego odporność na awarie,
- **Ułatwione wdrażanie i rozwój systemu** - poszczególne mikroservisy mogą być rozwijane, testowane oraz wdrażane niezależnie od siebie. Takie podejście upraszcza proces CI/CD, a także umożliwia równoległą pracę nad wieloma komponentami systemu,
- **Lepsze dopasowanie do skalowalności danych i baz danych** – mikroservisy mogą korzystać z różnych typów baz danych w zależności od potrzeb konkretnej domeny, co pozwala lepiej dopasować model danych do problemu.

Jednakże w takim podejściu architektonicznym również znajdziemy wady i wyzwania jakie się z tym wiążą. Niezależne skalowanie, wdrażanie jak i wersjonowanie usług powinno odbywać się z dużą ostrożnością. Usunięcie endpointu w serwisie bez wcześniejszego przeanalizowania jego użycia w innych serwisach może spowodować błędy po wdrożeniu, dlatego warto to dobrze analizować oraz stosować techniki, które temu zapobiegają. Pomimo licznych zalet architektura mikroservisowa wprowadza również dodatkową złożoność projektową i operacyjną.

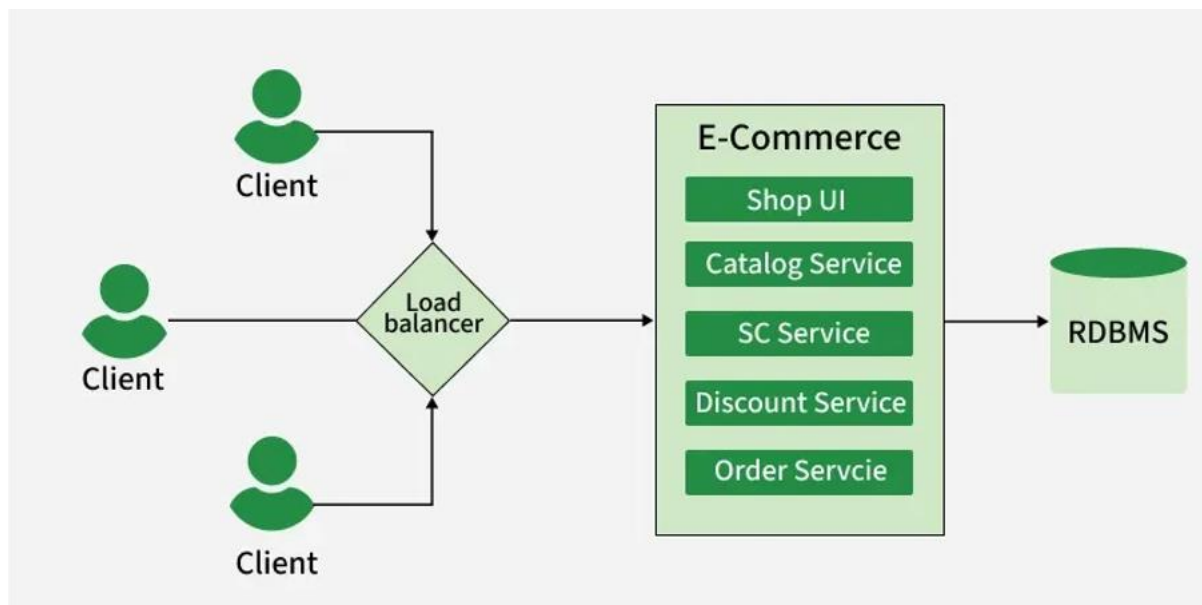
- **Komunikacja między serwisami** - konieczne jest zarządzanie komunikacją sieciową pomiędzy mikroservisami. Wymaga to odpowiedniego projektowania interfejsów API, obsługi błędów komunikacyjnych oraz zapewnienia niezawodności przesyłania danych,
- **Zarządzanie spójnością danych** - każdy mikroservis posiada własny kontekst danych, co utrudnia realizację transakcji obejmujących wiele usług jednocześnie. W projektowanym rozwiązaniu zastosowano operacje kompensujące zamiast klasycznych rozproszonych transakcji ACID,
- **Zwiększona złożoność infrastruktury** - architektura mikroservisowa wymaga stosowania dodatkowych mechanizmów związanych z monitorowaniem, logowaniem, orkiestracją usług oraz zarządzaniem konfiguracją środowiska. Powoduje to większą złożoność utrzymania całego systemu w porównaniu do architektury monolitycznej.

6.1.1 Porównanie pomiędzy mikroservisem i monolitem

Architektura monolityczna oraz mikroservisowa stanowią dwa różne podejścia do budowy systemów informatycznych. Różnią się one przede wszystkim sposobem organizacji kodu, modelu wdrażania aplikacji oraz zarządzania zależnościami pomiędzy komponentami. Wybór odpowiedniej architektury zależy od skali systemu, jego złożoności oraz wymagań dotyczących rozwoju i utrzymania co opisuje artykuł [45].

W podejściu monolitycznym wszystkie moduły jak i struktura aplikacji są częścią jednego wspólnego projektu. Oznacza to, że logika biznesowa, interfejsy oraz dostęp do danych są silnie powiązane i współdzielą ten sam model. W architekturze mikroservisowej system jest natomiast podzielony na niezależne usługi, z których każda odpowiada za określony obszar domenowy i może być rozwijana niezależnie. Odnośnie do skalowania systemu, który jest zbudowany według architektury monolitycznej, nawet jeśli jakaś drobna rzecz jest obciążana znacznie w danej chwili, obciążenie nałożone jest na cały projekt. W przypadku mikroservisów możliwe jest niezależne skalowanie poszczególnych usług, co pozwala lepiej dopasować zasoby infrastrukturalne do rzeczywistego obciążenia systemu.

W architekturze monolitycznej każda zmiana wymaga ponownego wdrożenia całej aplikacji, co może wydłużać proces dostarczania nowych funkcjonalności. Mikroserwisy umożliwiają niezależne wdrażanie poszczególnych komponentów, jednakże warto zwrócić uwagę na to, że w środowiskach produkcyjnych często zdarzają się sytuacje, gdzie w przypadku mikroserwisowej architektury zmiana lub dodanie nowej funkcjonalności dotyczy kilku z serwisów. Dlatego wdrażanie jest skalowalne natomiast jest o wiele bardziej złożone od wdrożeń monolitów.

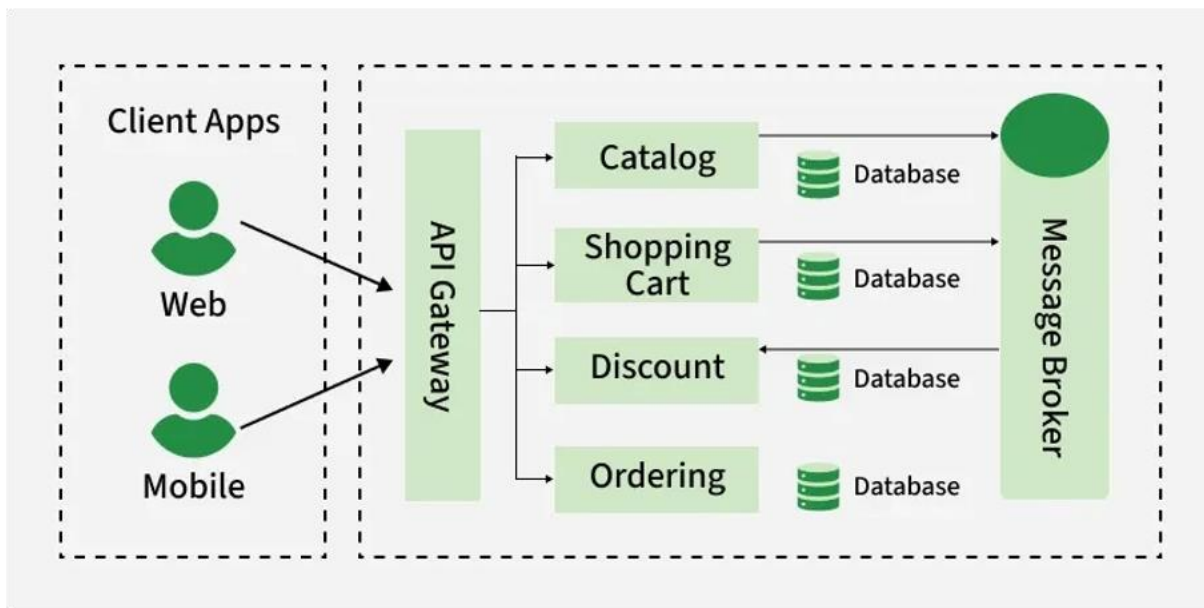


Rysunek 6: Przykład architektury monolitycznej, źródło: [46]

Wracając do złożoności, która jest o wiele większa dla systemów z architekturą mikroserwisową bądź ogólnie rozproszoną. Monolit jest prostszy w początkowej fazie rozwoju, jednak wraz ze wzrostem systemu jego utrzymanie staje się coraz trudniejsze, ze względu na dobudowywanie nowych modułów i operacji w jednym projekcie zazwyczaj nie uwzględniając istniejącej struktury. Mikroserwisy natomiast wprowadzają większą złożoność infrastrukturalną już od początku, wynikającą m.in. z konieczności zarządzania komunikacją sieciową, monitorowania usług oraz zapewnienia spójności danych co jest jedną z wad tej architektury, tym samym pozwalając na inne optymalizacje wydajności.

W architekturze monolitycznej zazwyczaj stosuje się jedną wspólną bazę danych, co upraszcza zarządzanie transakcjami. W mikroserwisach każda usługa posiada własną bazę danych, co zwiększa niezależność, ale utrudnia realizację transakcji rozproszonych i wymaga stosowania konkretnych podejść, które rozwiązują ten problem natomiast są bardziej złożone.

Podsumowując, architektura monolityczna jest prostsza w początkowej implementacji, jednak wraz ze wzrostem systemu może prowadzić do problemów związanych ze skalowalnością i utrzymaniem. Architektura mikroserwisowa, mimo większej złożoności operacyjnej, zapewnia większą elastyczność, skalowalność oraz niezależność od innych serwisów, dlatego została wybrana w projektowanym systemie, gdzie była potrzeba użycia architektury przystosowanej pod kątem rozłożenia obciążenia oraz optymalizacji procesów.



Rysunek 7: Przykład architektury mikroservisowej, źródło: [46]

6.1.2 Porównanie wzorców architektonicznych współczesnych systemów

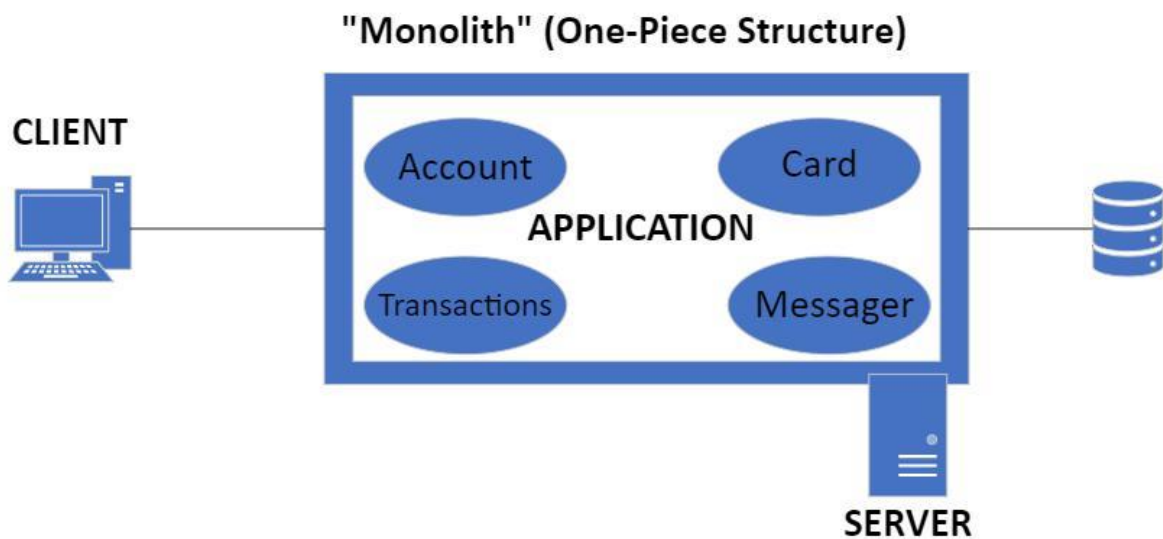
W dzisiejszych czasach na rynku jako architekci oraz inżynierzy spotkamy się z różnymi typami i modelami, według których budowane są systemy. Część wzorców utrzymywana jest od dekad, część była popularna oraz implementowana w systemach kilku lub kilkanaście lat temu, natomiast z roku na rok zauważamy, że te style architektoniczne się zmieniają. Powodem tego jest to, że firmy chcą podążać za rynkowymi trendami czasami nie wiedząc co dany wzorec za sobą niesie, jakie ma zalety, a jakie wady dla infrastruktury w jakiej dana firma się znajduje. McKinsey opisuje tzw. *antywzorce transformacji technologicznych*, gdzie technologie są wdrażane „bo są modne”, a nie dlatego, że rozwiązują problem o czym możemy więcej przeczytać w artykule [47].

Tak jak w życiu, każda dzisiaj podjęta decyzja wpływa na naszą przyszłość tak i również, każdy ze wzorców wpływa na dalsze kwestie związane z wydajnością systemu które dopiero po czasie mogą się pojawić. Wybierając model warto więc zapoznać się wcześniej z częścią z nich i zobaczyć w czym konkretny wzorec może nam zaszkodzić, a w czym pomóc. Poniżej przedstawiamy kilka z nich, aby móc je między sobą porównać i zobaczyć ich charakterystykę.

Architektura monolityczna

Architektura monolityczna jest najprostszym podejściem do budowy systemów informatycznych. W tym modelu cała aplikacja tworzona jest jako jeden spójny projekt, w którym wszystkie komponenty - interfejs użytkownika, logika biznesowa oraz warstwa dostępu do danych są ze sobą ściśle powiązane i działają w ramach jednego procesu uruchomieniowego.

Tego typu architektura jest często stosowana w początkowych fazach rozwoju systemów, ponieważ pozwala na szybkie tworzenie i wdrażanie aplikacji bez konieczności projektowania złożonej infrastruktury komunikacyjnej.



Rysunek 8: Architektura monolityczna, źródło: [48]

Zalety:

- prostszy proces wdrażania i konfiguracji,
- łatwiejsze testowanie całego systemu,
- mniejsze wymagania infrastrukturalne,
- dobra wydajność komunikacji wewnętrznej między modułami.

Wady:

- trudniejsza skalowalność dużych systemów,
- problematyczne wdrażanie zmian w rozbudowanych aplikacjach,
- wysoka zależność pomiędzy modułami,
- awaria jednego elementu może wpłynąć na działanie całego systemu.

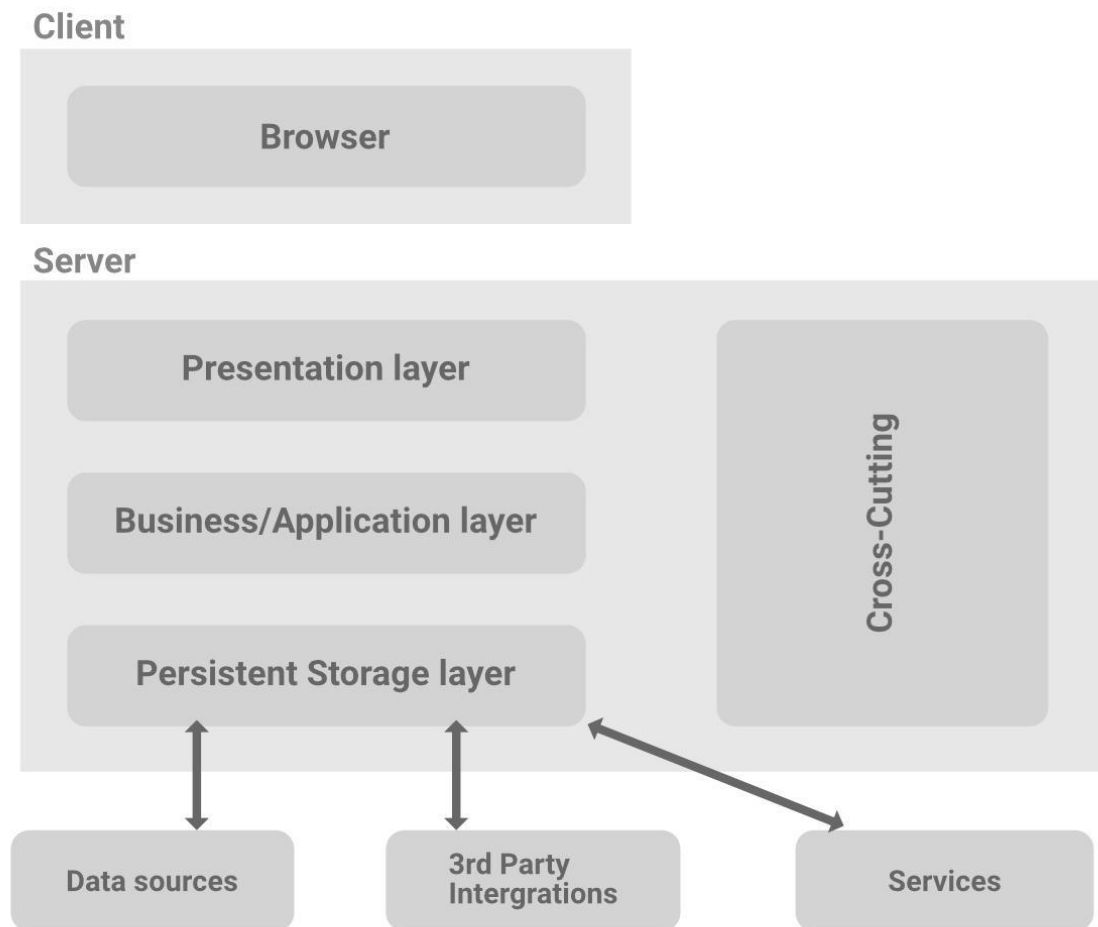
Architektura monolityczna sprawdza się głównie w małych i średnich projektach, gdzie priorytetem jest szybkie wdrożenie i prostota utrzymania.

Architektura warstwowa

Architektura warstwowa opiera się na podziale systemu na logiczne warstwy, z których każda odpowiada za inne zadania. Najczęściej wyróżnia się warstwę prezentacji (interfejs użytkownika), warstwę logiki biznesowej oraz warstwę dostępu do danych, czyli klasyczną architekturę *Model-View-Controller*.

Główną zaletą tego podejścia jest przejrzystość struktury oraz łatwość utrzymania kodu. Każda warstwa ma jasno określoną odpowiedzialność, co ułatwia zarówno rozwój, jak i testowanie aplikacji. Dzięki temu architektura ta jest często stosowana w systemach biznesowych, systemach ERP oraz klasycznych aplikacjach webowych.

Web Application Architecture



Rysunek 9: Przykład architektury wielowarstwowej, źródło: [49]

Zalety:

- przejrzysta organizacja kodu,
- łatwiejsze utrzymanie i rozwój aplikacji,
- możliwość niezależnego rozwijania poszczególnych warstw,
- dobra czytelność projektu dla zespołu programistycznego.

Wady:

- dodatkowe narzuty komunikacyjne pomiędzy warstwami,
- ograniczona elastyczność przy bardzo dużych systemach,
- możliwość powstawania nadmiernie rozbudowanej logiki biznesowej,
- trudniejsza optymalizacja wydajności.

Architektura mikroserwisowa

Architektura mikroserwisowa stanowi przeciwieństwo monolitu. System jest tutaj podzielony na wiele małych, niezależnych usług, z których każda odpowiada za określoną funkcjonalność biznesową. Usługi te komunikują się ze sobą najczęściej za pomocą API albo systemów kolejkowych.

Najważniejszą cechą tego podejścia jest niezależność komponentów - każda usługa może być rozwijana, testowana i wdrażana osobno. Dzięki temu zespoły programistyczne mogą pracować równolegle nad różnymi częściami systemu, co znacząco przyspiesza rozwój dużych aplikacji.

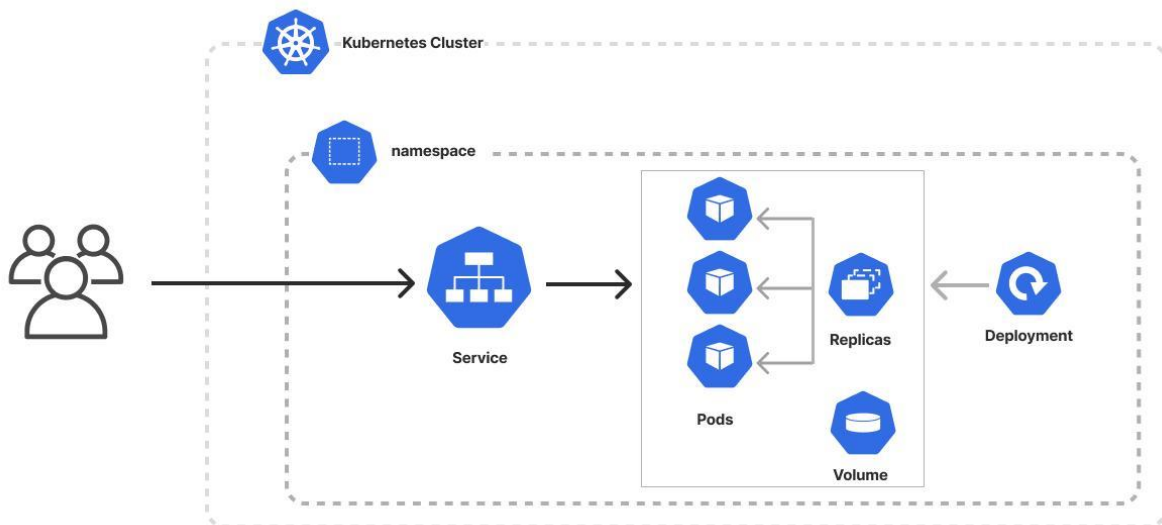
Mikroserwisy zapewniają również lepszą skalowalność, ponieważ możliwe jest skalowanie tylko tych elementów systemu, które faktycznie tego wymagają. Z drugiej strony, wprowadzenie tej architektury wiąże się z dużą złożonością operacyjną, koniecznością stosowania w wielu przypadkach narzędzi automatyzujących DevOps oraz trudniejszym zarządzaniem spójnością danych.

Zalety:

- wysoka skalowalność systemu,
- możliwość niezależnego wdrażania usług,
- większa odporność na awarie,
- łatwiejszy rozwój dużych projektów przez wiele zespołów,
- możliwość wykorzystania różnych technologii dla poszczególnych usług.

Wady:

- większa złożoność infrastruktury,
- trudniejsze zarządzanie komunikacją między usługami,
- bardziej skomplikowane monitorowanie i debugowanie,
- wyższe koszty utrzymania środowiska.



Rysunek 10: Architektura mikroserwisowa, źródło: [50]

Architektura mikroserwisowa jest obecnie szeroko stosowana w dużych systemach chmurowych, platformach e-commerce oraz aplikacjach o dużej liczbie użytkowników i procesowania informacji.

Architektura oparta na zdarzeniach (Event-Driven Architecture)

W architekturze opartej na zdarzeniach komponenty komunikują się poprzez przesyłanie zdarzeń. System reaguje na określone akcje użytkownika lub zmiany stanu aplikacji.

Zalety:

- wysoka elastyczność systemu,
- możliwość przetwarzania danych w czasie rzeczywistym,
- łatwiejsza integracja wielu usług,
- dobra skalowalność dla systemów rozproszonych.

Wady:

- trudniejsze śledzenie przepływu danych,
- większa złożoność debugowania,
- problemy z zachowaniem spójności danych,
- bardziej wymagające projektowanie komunikacji.

Ten wzorzec znajduje zastosowanie m.in. w systemach IoT⁹⁵, aplikacjach finansowych, platformach streamingowych oraz systemach analitycznych wymagających natychmiastowych reakcji.

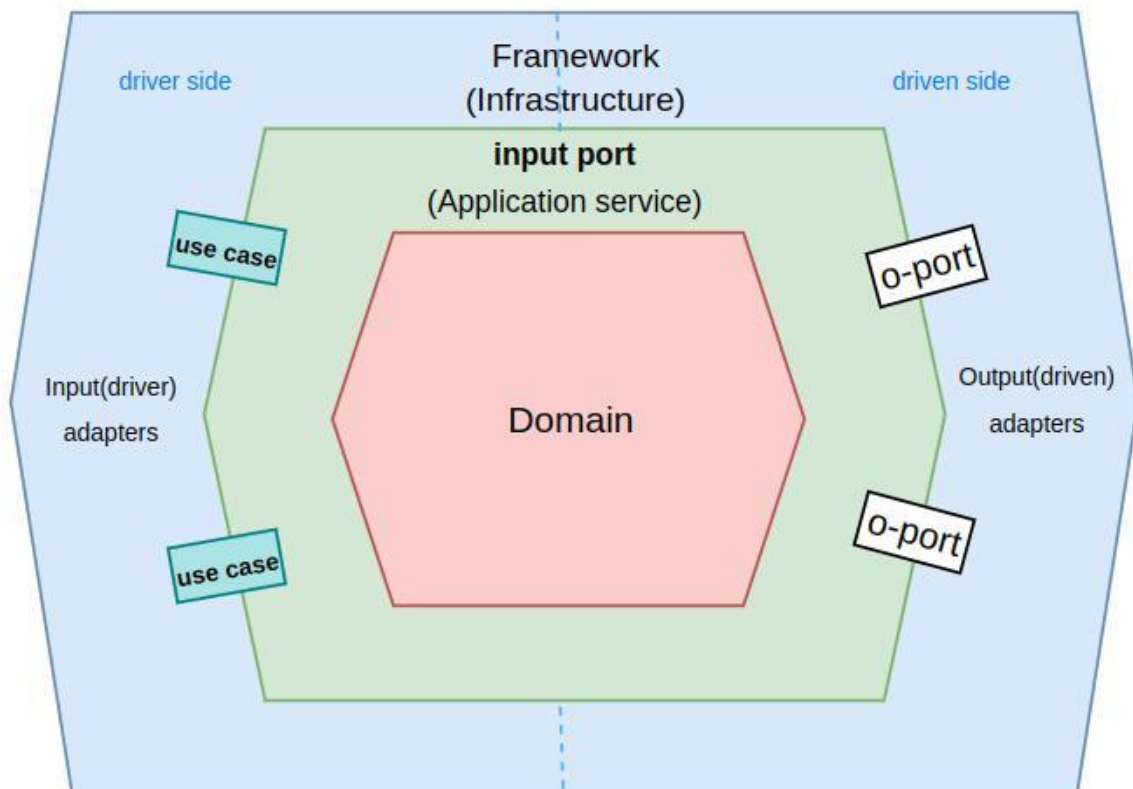
⁹⁵ Koncepcja sieci połączonych urządzeń fizycznych, które zbierają i wymieniają dane przez Internet. IoT umożliwia automatyzację, monitorowanie i analizę w czasie rzeczywistym

Architektura heksagonalna (Hexagonal Architecture / Ports & Adapters)

Architektura heksagonalna, znana również jako *Porty i adaptery*, została stworzona w celu poprawy testowalności oraz niezależności logiki biznesowej od technologii zewnętrznych.

W tym podejściu centralnym elementem systemu jest *rdzeń aplikacji (domain/core)*, który zawiera logikę biznesową. Wokół niego znajdują się *porty*, czyli interfejsy definiujące sposób komunikacji z systemem. Z kolei *adaptery* są implementacjami tych portów i odpowiadają za integrację z zewnętrznymi systemami, np. bazą danych, REST API czy interfejsem użytkownika.

Najważniejszą ideą tej architektury jest to, że logika biznesowa nie zależy od technologii, lecz odwrotnie - to technologie są „podpinane” do systemu. Dzięki temu można łatwo wymieniać np. bazę danych bez ingerencji w rdzeń aplikacji.



Rysunek 11: Architektura heksagonalna, źródło: [51]

Zalety:

- bardzo dobra testowalność (można testować logikę bez infrastruktury),
- wysoka niezależność od technologii,
- łatwość zmiany komponentów zewnętrznych,
- czysta separacja logiki biznesowej.

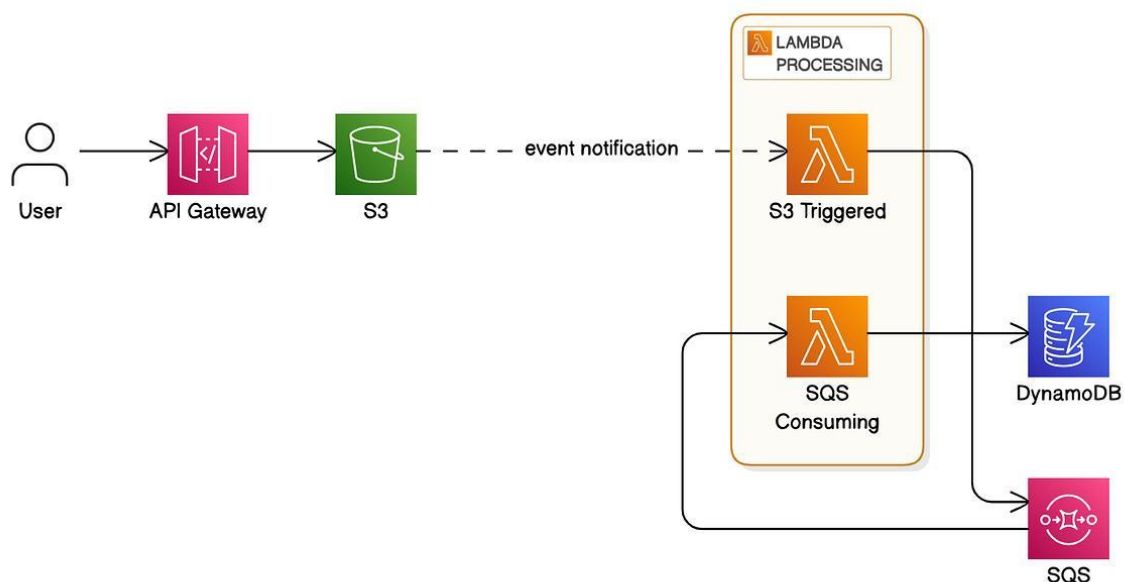
Wady:

- większa liczba abstrakcji,
- wyższa złożoność projektu,
- trudniejsza nauka dla początkujących,
- więcej kodu do napisania i utrzymywania (interfejsy, adaptory).

Architektura serverless (bezserwerowa)

Architektura **serverless** to model, w którym aplikacja nie działa na stałych serwerach zarządzanych przez programistę, lecz w formie funkcji uruchamianych na żądanie. Najczęściej realizowana jest w modelu **FaaS**⁹⁶, np. AWS Lambda, Azure Functions lub Google Cloud Functions.

W tym podejściu programista nie zarządza infrastrukturą - serwerami, skalowaniem czy utrzymaniem systemu. Zamiast tego dostarcza funkcje, które są uruchamiane automatycznie w odpowiedzi na zdarzenia, takie jak żądanie HTTP, zapis w bazie danych czy komunikat w kolejce. Serverless jest często łączony z architekturą event-driven i mikroserwisami, ponieważ naturalnie wspiera podejście oparte na zdarzeniach.



Rysunek 12: Architektura serverless, źródło: [52]

Zalety:

- brak zarządzania serwerami,
- automatyczne skalowanie,
- płatność tylko za użycie,

⁹⁶ Model przetwarzania w chmurze, w którym dostawca chmury uruchamia i zarządza pojedynczymi funkcjami aplikacji w odpowiedzi na zdarzenia. Programista nie musi zarządzać serwerami ani infrastrukturą - płaci się tylko za czas wykonania funkcji

- szybkie wdrażanie funkcji.

Wady:

- występowanie „cold start”⁹⁷,
- ograniczona kontrola nad środowiskiem,
- trudniejsze debugowanie rozproszonego systemu,
- zależność od dostawcy chmury (*vendor lock-in*).

Tabela 2: Porównanie wzorców projektowych

Cecha	Monolit	Warstwowa	Mikroserwisy	EDA	Heksagonalna	Serverless
Złożoność początkowa	Niska	Niska/Średnia	Wysoka	Wysoka	Średnia	Niska
Złożoność operacyjna	Niska	Niska	Bardzo wysoka	Wysoka	Średnia	Niska/Średnia
Skalowanie	Cała aplikacja	Cała aplikacja	Każdy serwis osobno	Każdy konsument osobno	Zależy od wdrożenia	Automatycznie
Łatwość wdrożenia	Bardzo łatwa	Bardzo łatwa	Trudna	Trudna	Średnia	Bardzo łatwa
Koszt utrzymania	Niski	Niski	Wysoki	Wysoki	Średni	Zależy od rozłożenia
Najlepsze zastosowanie	MVP, małe projekty	Klasyczne aplikacje biznesowe	Duże systemy	Systemy czasu rzeczywistego	Projekty stawiające na testowalność i modułowość	Nieregularne obciążenie, integracje

Nowoczesne systemy coraz częściej wykraczają poza klasyczne modele architektoniczne. Architektura heksagonalna koncentruje się na czystości logiki biznesowej i jej niezależności od technologii, natomiast serverless zmienia sposób myślenia o infrastrukturze, przenosząc odpowiedzialność za jej utrzymanie na dostawcę chmury. W praktyce bardzo często łączy się różne podejścia - np. mikroserwisy wdrażane w modelu serverless lub architekturę heksagonalną wewnątrz pojedynczych usług.

⁹⁷ Opóźnienie występujące przy pierwszym uruchomieniu funkcji serverless (lub po dłuższej przerwie w jej użyciu), gdy środowisko wykonawcze musi zostać dopiero „rozgrzane” (np. uruchomienie kontenera, inicjalizacja runtime, załadowanie kodu)

6.2. Podział na poszczególne mikroserwisy w projekcie

Architektura opracowanego systemu, tak jak to zostało wspomniane w poprzednim podrozdziale - jest zaimplementowana we wzorcu mikroserwisowym. Oznacza to podział na poszczególne niezależne moduły oraz domeny odpowiedzialne za logikę biznesową jak i architektoniczną. Żeby móc zobrazować ten podział, który pomoże w zrozumieniu połączeń i złożoności, poniżej zostały przedstawione moduły z opisami oraz konkretne części systemu współpracujące między sobą. Poszczególne moduły mikroserwisowe komunikują się ze sobą w sposób kontrolowany, najczęściej poprzez API lub mechanizmy komunikacji asynchronicznej, co zapewnia spójność danych oraz stabilność całej architektury.

Warstwa odpowiedzialna za bezpośredni kontakt z klientem składa się z dwóch oddzielnych domen biznesowych - jednej dla najemcy oraz drugiej dla zarządcy. Są to dwa serwisy frontendowe ustawione na oddzielnych portach, które służą jako dwa odrębne panele dla konkretnych operacji i ról systemowych. Każdy z nich posiada dedykowany interfejs użytkownika dostosowany do potrzeb danej grupy użytkowników, co pozwala na lepszą ergonomię pracy oraz ograniczenie dostępu do funkcjonalności nieprzeznaczonych dla danej innych nieautoryzowanych ról.

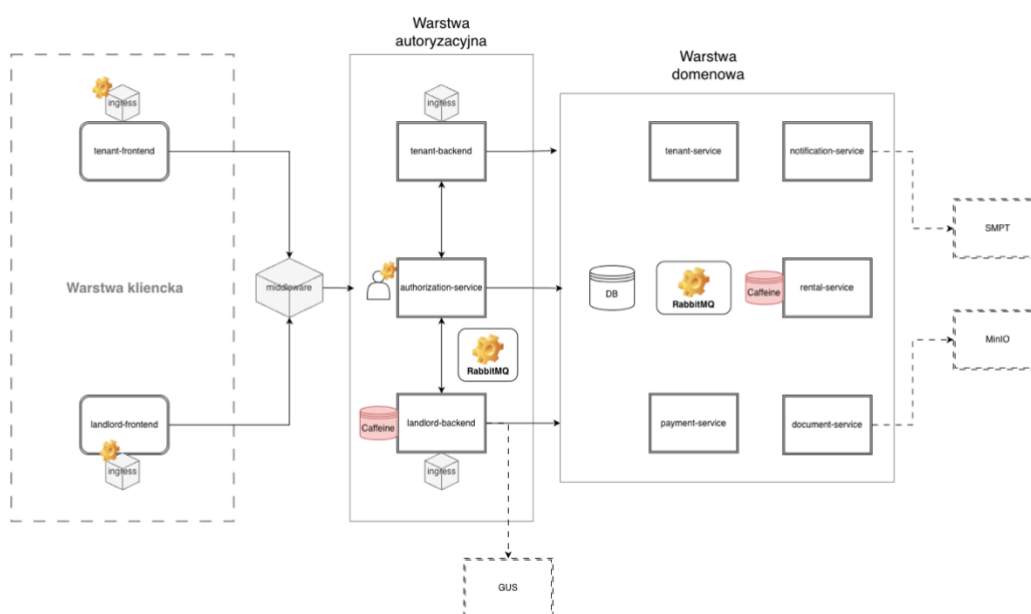
- Panel wynajmującego (Angular) - wystawiony na porcie 4200,
- Panel najemcy (Angular) - wystawiony na porcie 7200

Serwisy frontendowe komunikują się z warstwą serwerową za pośrednictwem dedykowanych backendów pełniących rolę bram API (*Backend for Frontend* - BFF). Każdy z backendów obsługuje wyłącznie jedną domenę aplikacji frontendowej i odpowiada za weryfikację połączeń, autoryzację użytkowników oraz kontrolę i przetwarzanie przekazywanych danych. Dzięki temu możliwe jest dostosowanie logiki i zakresu danych do specyfiki danego interfejsu użytkownika, przy jednoczesnej separacji frontendów od wewnętrznej architektury mikroserwisowej.

- **landlord-backend (Spring Boot)** - Backend dedykowany aplikacji zarządcy (landlord). Pełni rolę centralnej bramy API dla panelu administracyjno-zarządczego. Odpowiada za:
 - agregację danych z wielu mikroserwisów,
 - transformację i dostosowanie danych do potrzeb interfejsu użytkownika,
 - obsługę procesów biznesowych związanych z zarządzaniem nieruchomościami, najemcami, płatnościami i zgłoszeniami,
 - autoryzację i walidację żądań pochodzących z aplikacji frontendowej,
 - ukrycie wewnętrznej struktury systemu przed frontendem.
- **tenant-backend (Spring Boot)** - Backend dedykowany aplikacji najemcy (tenant). Udostępnia wyłącznie funkcjonalności i dane dostępne dla użytkownika końcowego - najemcy. Odpowiada za:
 - bezpieczny dostęp do danych związanych z umową, płatnościami i zgłoszeniami,
 - egzekwowanie uprawnień użytkownika i ograniczenie dostępu wyłącznie do jego zasobów,
 - uproszczenie komunikacji frontendowej poprzez udostępnienie jednego punktu wejścia do systemu.

Następnie dzięki tym dedykowanym serwisom połączenie przekazywane jest dalej do serwisów wewnętrznych, w których wykonuje się już logika biznesowa. Te serwisy nie są dostępne przez aplikacje z zewnątrz, natomiast mogą komunikować się ze sobą z zachowaniem odpowiedniego podziału logiki domenowej. Podział tych serwisów wygląda następująco:

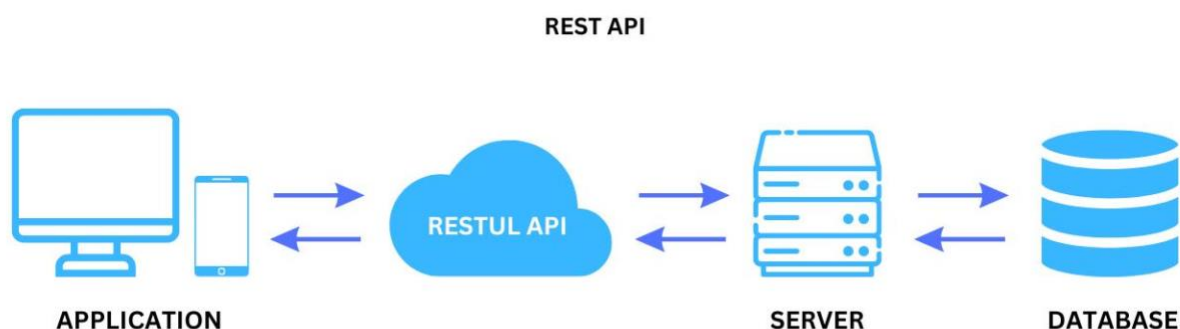
- **authorization-service (Spring Boot)** - odpowiada za uwierzytelnienie, wystawianie kluczy uwierzytelniających oraz zarządzanie kontami i uprawnieniami użytkowników,
- **rental-service (Spring Boot)** - rdzeń systemu, zarządza lokalami, umowami najmu, opłatami, harmonogramem zadań cyklicznych oraz orkiestruje zdarzenia domenowe,
- **tenant-service (Spring Boot)** - przechowuje dane najemców i zarządza ich cyklem życia,
- **payment-service (.NET)** - obsługuje transakcje płatnicze, generuje faktury i publikuje zdarzenia związane z płatnościami oraz ich potwierdzeniem,
- **document-service (.NET)** - serwis napisany w technologii .NET odpowiedzialny za generowanie plików PDF na podstawie szablonów i przechowywanie ich w obiektywnym magazynie plików MinIO,
- **notification-service (Spring Boot)** - nasłuchuje na zdarzenia w kolejce i wysyła wiadomości e-mail za pośrednictwem protokołu SMTP, korzystając z silnika *Thymeleaf*.



Rysunek 13: Diagram architektury systemu

6.3. Architektura REST API

Architektura REST API stanowi jeden z kluczowych elementów współczesnych systemów informatycznych, umożliwiając komunikację pomiędzy różnymi komponentami aplikacji w sposób ustandaryzowany, skalowalny oraz niezależny od wykorzystywanych technologii. REST jest stylem architektonicznym opartym na protokole HTTP, którego głównym założeniem jest wymiana zasobów pomiędzy klientem, a serwerem przy użyciu jasno określonych metod komunikacji. System, któremu poświęcona jest ta praca posiada zaimplementowane dwa wzorce komunikacji w zależności od charakteru operacji. Głównym z nich jest oczywiście REST API.

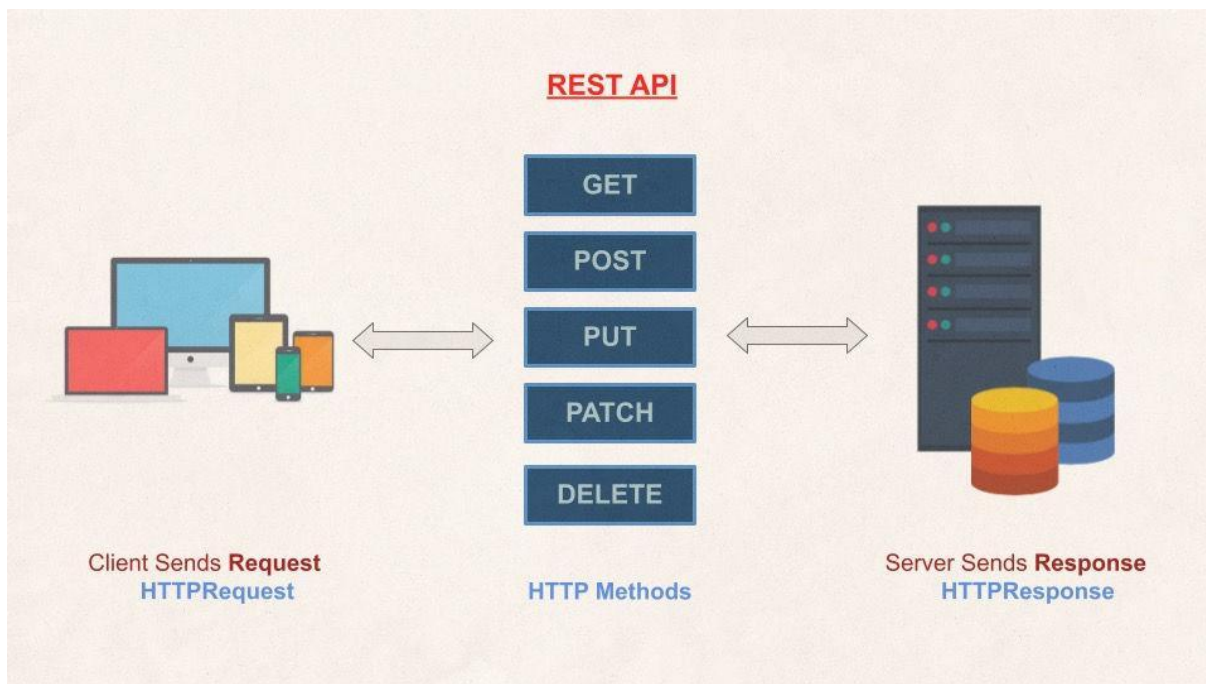


Rysunek 14: Diagram REST API, źródło: [53]

W praktyce REST API pełni rolę warstwy pośredniczącej pomiędzy frontendem aplikacji, a logiką biznesową oraz bazą danych. Dzięki temu możliwe jest oddzielenie warstwy prezentacji od warstwy przetwarzania danych, co znacząco zwiększa modularność, elastyczność oraz łatwość dalszego rozwoju systemu. Komunikacja odbywa się zazwyczaj z wykorzystaniem formatu JSON, który zapewnia lekkość przesyłanych danych i wysoką kompatybilność pomiędzy różnymi środowiskami programistycznymi. Natomiast oprócz warstwy pośredniczącej pomiędzy aplikacją kliencką, a serwerową jest też główną szyną potrzebna do komunikacji pomiędzy serwisami w naszym ekosystemie, a innymi integracjami z zewnętrznymi usługami.

Istotną cechą architektury REST jest bezstanowość⁹⁸, oznaczająca, że każde żądanie klienta zawiera komplet informacji niezbędnych do jego obsługi [54]. Takie podejście upraszcza zarządzanie systemem oraz zwiększa jego skalowalność. REST API wykorzystuje również standardowe metody HTTP, takie jak GET, POST, PUT czy DELETE, które odpowiadają podstawowym operacjom wykonywanym na zasobach [55].

⁹⁸ Serwer nie przechowuje informacji o stanie sesji klienta pomiędzy kolejnymi żądaniami - każde żądanie jest niezależne



Rysunek 15: Przykład dostępnych metod dla protokołu HTTP

6.3.1 Komunikacja synchroniczna kontra asynchroniczna

System posiada zaimplementowane dwa typy komunikacji, czyli synchroniczną oraz asynchroniczną. Komunikacja odbywa się wewnętrznie, aby do przesyłania danych pomiędzy mikroserwisami oraz zewnętrznie do wysyłania i odbierania danych z aplikacji klienckich do warstwy serwerowej.

- Komunikacja synchroniczna (REST/HTTP) - stosowana wtedy, gdy klient wysyłający żądanie oczekuje na natychmiastową odpowiedź od serwera. Oznacza to, że dalsze przetwarzanie po stronie klienta jest wstrzymane do momentu otrzymania wyniku. Takie podejście jest typowe dla klasycznych wywołań REST API opartych o protokół HTTP, gdzie każde żądanie kończy się odpowiedzią w ramach tego samego połączenia. Rozwiązanie to jest proste w implementacji i intuicyjne, jednak może prowadzić do spadku wydajności w przypadku operacji długotrwałych lub obciążających system,
- Komunikacja asynchroniczna (RabbitMQ) - stosowana dla operacji, gdy klient nie oczekuje natychmiastowej odpowiedzi na wysłane żądanie. Zamiast tego otrzymuje jedynie potwierdzenie przyjęcia zadania do realizacji, a właściwy wynik dostarczany jest w późniejszym czasie, np. poprzez *webhook*, kolejkę wiadomości lub dodatkowe zapytanie. Podejście to zwiększa skalowalność systemu oraz pozwala na efektywne przetwarzanie zadań o dużej złożoności, które nie muszą być realizowane w czasie rzeczywistym.

W praktyce nowoczesne systemy często łączą oba podejścia. Komunikacja synchroniczna stosowana jest tam, gdzie wymagany jest natychmiastowy wynik, natomiast asynchroniczna wykorzystywana jest w przypadku operacji długotrwałych, takich jak przetwarzanie plików, wysyłka powiadomień czy integracje z zewnętrznymi usługami [56]. W poniższych listingach

możemy zobaczyć zaimplementowane porównanie klientów oraz odbiorców obsługi synchronicznej oraz asynchronicznej.

Wywołanie klienta serwisu rental-service z serwisu pośredniczącego landlord-backend z aktywacją konta zarządcy.

Listing 8: Implementacja klienta REST oraz jego użycie

landlord-backend:

```
void activate(@NonNull String userId) {
    Log.info("Activate landlord account by user id {}", userId);
    long start = System.currentTimeMillis();
    try {
        restClient.post()
            .uri(UriComponentsBuilder.fromUriString(rentalServiceUrl)
                .pathSegment("landlords", "activate")
                .queryParams("userId", userId)
                .build()
                .toUriString())
            .retrieve()
            .toBodilessEntity();
    } catch (HttpClientErrorException | HttpServerErrorException e) {
        Log.error(
            "Unable to activate landlord account in [{}] ms - [{}]",
            System.currentTimeMillis() - start,
            e.getResponseBodyAsString(), e);
        throw e;
    } catch (Exception e) {
        Log.error("Unable to activate landlord account in [{}] ms", S
            ystem.currentTimeMillis() - start, e);
        throw e;
    }
}
```

rental-service:

```
@PostMapping(value = "/activate", params = {"userId"})
public ResponseEntity<Void> activate(@RequestParam String userId) {
    try {
        service.activate(new LandlordUserAdapter(userId));
        return ResponseEntity.ok().build();
    } catch (Exception e) {
        Log.error(e.getMessage(), e);
        throw new ResponseStatusException(
            HttpStatus.INTERNAL_SERVER_ERROR,
            "Unable to activate landlord"
        );
    }
}
```

Natomiast implementacja komunikacji asynchronicznej zaimplementowanej dzięki brokerom wiadomości oraz zdarzeniom jako *konsumer* takiego zdarzenia przychodzącego wygląda następująco jak na listingu 9 i 10.

Listing 9: Emisja wewnętrznego zdarzenia po poprawnym zapisie do bazy danych

```
// RentalServiceImpl.java - publikacja zdarzenia domenowego
RentalEntity savedEntity = repository.save(entity);
eventPublisher.publishEvent(new RentalCreatedEvent(
    factory.build(savedEntity, rental.getFees()), tenant, property, landlord
));
```

Listing 10: Odebranie zdarzenia i obsługa dalszej części

```
// RentalEventListener.java - obsługa po COMMIT transakcji
@TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
public void handleRentalInitializationSuccess(@NonNull RentalCreatedEvent event) {
    emailService.send(...); // - RabbitMQ [queue: emails]
    documentService.send(...); // - RabbitMQ [queue: documents]
}
```

Zdarzenia pomiędzy serwisami wysyłane są za pomocą klasy RabbitTemplate, tak jak zostało to pokazane na listingu 11.

Listing 11: Implementacja klasy RabbitTemplate odpowiedzialnej za wysyłkę zdarzeń

```
// EmailAMQPClient.java
void send(@NonNull EmailQueueMessageDTO message) {
    rabbitTemplate.convertAndSend(
        EmailAMQPKeys.EXCHANGE_EMAILS, EmailAMQPKeys.QUEUE_EMAILS, message);
}
```

Konsumenci zdarzeń deklarowani są przez @RabbitListener, który obsługuje zarówno pojedyncze zdarzenia jak i paczki.

Listing 12: Metoda nasłuchująca na zdarzenie płatności

```
// PaymentAMQPListener.java
@RabbitListener(queues = PaymentEvent.SUCCESS_QUEUE_NAME)
public void handleRentalPaymentSuccessEvent(PaymentEvent event){ ... }
```

Listing 13: Metoda nasłuchująca na zdarzenie związane z mailem

```
// EmailAMQPListener.java (notification service)
@RabbitListener(queues = EmailAMQPKeys.QUEUE_EMAILS)
public void receiveMessage(EmailQueueMessageDTO message) {
    emailService.sendEmail(message);
}
```

}

Zalety i wady komunikacji synchronicznej i asynchronicznej

Komunikacja synchroniczna charakteryzuje się prostotą implementacji oraz łatwością zrozumienia działania, ponieważ każde żądanie kończy się natychmiastową odpowiedzią. Dzięki temu jest intuicyjna w użyciu i dobrze sprawdza się w przypadku operacji wymagających szybkiej reakcji systemu. Do jej wad należy jednak zaliczyć blokowanie procesu klienta do momentu otrzymania odpowiedzi, co może prowadzić do spadku wydajności oraz problemów przy długotrwałych operacjach lub dużym obciążeniu systemu.

Z kolei komunikacja asynchroniczna zapewnia większą skalowalność oraz lepsze wykorzystanie zasobów systemowych, ponieważ klient nie musi oczekiwać na natychmiastową odpowiedź. Umożliwia to efektywne przetwarzanie zadań w tle oraz obsługę większej liczby równoczesnych operacji. Wadą tego podejścia jest natomiast większa złożoność implementacyjna oraz trudniejsze zarządzanie przepływem danych, ponieważ wymaga dodatkowych mechanizmów, takich jak kolejki wiadomości, systemy powiadomień lub *webhooks*.

6.4. Architektura tokenów uwierzytelniających

Podczas implementacji, bezpieczeństwo systemu oraz użytkownika jest również bardzo ważne jak sama architektura. Dlatego do całości ekosystemu zostały zaimplementowane moduły pozwalające na autentykację oraz autoryzację użytkowników jak i danych wchodzący do serwera. Ze względu na *customową* implementację *OAuth 2.0* w systemie, który służy jako łącznik całego bezpieczeństwa serwisów, musiał zostać utworzony własny moduł przechowywania wrażliwych danych, moduł odpowiedzialny za weryfikację oraz wystawianie uwierzytelnienia - służący w systemie jako provider⁹⁹. Za główną część bezpieczeństwa odpowiada *authorization-service* implementowany przy pomocy *Spring Boot OAuth2 Authorization Server* oraz dwa dedykowane serwisy przez które przechodzi ruch z aplikacji klienckich. *Authorization-service* jest serwisem generycznym, co przekłada się na możliwość zmiany providera - z własnego na Keycloak bądź Google oraz dostosowanie konkretnych operacji pozwalających na korzystanie z tych platform do weryfikacji użytkowników.

Moduły uwierzytelniające opierają się na kluczach JWT szyfrowanych certyfikatem wewnętrznym, natomiast klucz publiczny udostępniony jest poprzez endpoint JWKS konkretnym serwisom pośredniczącym z aplikacjami klienckimi, dzięki któremu weryfikują klucze JWT przekazywane w requestach. W systemie istnieją dwa typy kluczy uwierzytelniających, gdzie jeden służy jako token główny pozwalający na korzystanie z aplikacji, i drugi który wygasa po 5 minutach i pozwala tylko na zmianę hasła podczas tworzenia konta użytkownika. Każdy z kluczy systemowych zawiera informacje o nazwie użytkownika, kontekstu z jakiego dany użytkownik pochodzi oraz jego zasadach dostępu. Mechanizm ten wyklucza konieczność stosowania haseł tymczasowych tworzonych przez administratora systemu oraz został przedstawiony na listingu 14.

⁹⁹ Zewnętrzny system bądź usługa, dostawca usług, zasobów lub infrastruktury, np. dostawca usług internetowych, chmurowych lub telekomunikacyjnych

Listing 14: Implementacja tokenów pośredniczących w autoryzacji

```
private static final int TTL_SECONDS = 3600;
private static final int TEMPORARY_TTL_SECONDS = 300;
// Token standardowy - po zalogowaniu
@Override
public String generateAccessToken(
    @NonNull UserDetailsAdapter userDetails,
    @NonNull String contextId
) {
    String scope = userDetails.getAuthorities().stream()
        .map(GrantedAuthority::getAuthority)
        .collect(Collectors.joining(" "));

    Instant now = Instant.now();
    Instant exp = now.plus(TTL_SECONDS, ChronoUnit.SECONDS);
    JwtClaimsSet claims = JwtClaimsSet.builder()
        .issuer(issuer)
        .issuedAt(now)
        .expiresAt(exp)
        .audience(List.of("SYSTEM"))
        .subject(userDetails.getPublicId())
        .claim("username", userDetails.getUsername())
        .claim("contextId", contextId)
        .claim("realm", userDetails.getRealm())
        .claim("scope", scope)
        .build();
    JwsHeader headers = JwsHeader.with(SignatureAlgorithm.RS256)
        .keyId(kid)
        .build();

    return jwtEncoder.encode(JwtEncoderParameters
        .from(headers, claims))
        .getTokenValue();
}

// Token jednorazowy - wyłącznie do ustawienia hasła
@Override
public String generatePasswordSetupAccessToken(
    @NonNull UserDetailsAdapter userDetails
) {
    Instant now = Instant.now();
    Instant exp = now.plus(TEMPORARY_TTL_SECONDS, ChronoUnit.SECONDS);

    JwtClaimsSet claims = JwtClaimsSet.builder()
        .issuer(issuer)
        .issuedAt(now)
        .audience(List.of("TEMPORARY"))
        .expiresAt(exp)
        .subject(userDetails.getPublicId())
        .claim("username", userDetails.getUsername())
        .claim("realm", userDetails.getRealm())
        .claim("scope", "password:set")
        .build();
    JwsHeader headers = JwsHeader.with(SignatureAlgorithm.RS256)
        .keyId(kid)
```

```

        .build();

    return jwtEncoder.encode(JwtEncoderParameters
        .from(headers, claims))
        .getTokenValue();
}

```

6.4.1 Klucze asymetryczne do podpisywania

Każdy z tokenów JWT przed wyjściem do użytkownika zostaje podpisany specjalnym certyfikatem, który zawiera klucz prywatny dostępny tylko dla serwisu autoryzacyjnego oraz klucz publiczny, który udostępniony jest poprzez specjalny endpoint dla dedykowanych serwisów pośredniczących z aplikacjami klienckimi. Dany certyfikat w systemie, któremu poświęcona jest praca został utworzony za pomocą wbudowanego generatora znajdującego się w systemach operacyjnych *OpenSSL*, ze względu na badawczy cel systemu oraz brak potrzeby zachowania prawdziwości takiego certyfikatu. W środowiskach produkcyjnych taki certyfikat musiałby być wystawiony przez specjalne centralne organy pośredniczące i uwzględniające ważność takich kluczy jak i również rotację certyfikatów w systemie co jakiś określony czas, żeby móc zachować bezpieczeństwo oraz minimalizować zagrożenia pochodzące z potencjalnego wyciągnięcia takiego certyfikatu i sfałszowania klucza JWT. Przykład tworzenia certyfikatu do celów badawczych został pokazany na listingu 15.

Listing 15: Implementacja tworzenia certyfikatu

```

openssl genrsa -out keystore/jwt-private.pem 3072
openssl req -new -x509 \
    -key keystore/jwt-private.pem \
    -out keystore/jwt-cert.pem \
    -days 3650 \
    -subj "/CN=authorization-service"

```

6.4.2 OAuth 2.0

OAuth 2.0 jest otwartym standardem autoryzacji, umożliwiającym aplikacjom uzyskanie ograniczonego dostępu do zasobów użytkownika bez konieczności przekazywania jego danych uwierzytelniających, takich jak login i hasło. Mechanizm ten opiera się na wydawaniu tokenów dostępowych, które określają zakres uprawnień oraz czas ich ważności. RFC 6749 stanowi oficjalną specyfikację protokołu *OAuth 2.0* opracowaną przez Dick Hardt’a oraz społeczność *OAuth Working Group* [57]. Dokument ten definiuje role uczestników, przepływy autoryzacji oraz sposób wydawania i weryfikacji tokenów dostępu.

W modelu *OAuth 2.0* użytkownik może wyrazić zgodę na dostęp do wybranych zasobów przez zewnętrzną aplikację, przy zachowaniu kontroli nad zakresem udzielonych uprawnień. Rozwiązanie to jest powszechnie stosowane w integracjach z interfejsami API oraz podczas autoryzacji dostępu do usług internetowych. Dzięki rozdzieleniu procesu autoryzacji od uwierzytelniania *OAuth 2.0* zwiększa bezpieczeństwo komunikacji pomiędzy aplikacjami i ogranicza ryzyko ujawnienia danych logowania. W systemie ten model autoryzacji został wykorzystany jedynie jako część łącząca pod spodem całe bezpieczeństwo, co pokazuje listing 16.

Listing 16: Konfiguracja podstawy OAuth 2.0

```
public SecurityFilterChain authorizationServerSecurityFilterChain(
    HttpSecurity http
) throws Exception {
    OAuth2AuthorizationServerConfigurer authorizationServer =
        new OAuth2AuthorizationServerConfigurer();
    authorizationServer.oidc(withDefaults());
    http
        .securityMatcher(authorizationServer.getEndpointsMatcher())
        .authorizeHttpRequests(auth -> auth.anyRequest().authenticated())
        .csrf(csrf -> csrf.ignoringRequestMatchers(
            authorizationServer.getEndpointsMatcher()))
        .with(authorizationServer, configurer ->
            configurer.oidc(withDefaults()));
    return http.build();
}
```

6.4.3 OAuth 2.0 vs klasyczna autoryzacja

W tradycyjnym modelu autoryzacji użytkownik przekazuje aplikacji swoje dane logowania, takie jak login i hasło, a aplikacja wykorzystuje je do uzyskania dostępu do określonych zasobów. Rozwiązanie to jest stosunkowo proste, jednak wiąże się z większym ryzykiem bezpieczeństwa, ponieważ dane uwierzytelniające są przekazywane podmiotowi trzeciemu i mogą zostać przechwycone lub niewłaściwie wykorzystane.

W przypadku *OAuth 2.0* użytkownik nie udostępnia aplikacji swoich danych logowania. Zamiast tego autoryzacja odbywa się za pośrednictwem serwera autoryzacyjnego, który po uzyskaniu zgody użytkownika wydaje token dostępu. Token ten określa zakres przyznanych uprawnień oraz czas ich ważności. Dzięki temu aplikacja otrzymuje dostęp wyłącznie do wybranych zasobów, bez konieczności przechowywania hasła użytkownika.

Zastosowanie *OAuth 2.0* zwiększa poziom bezpieczeństwa oraz umożliwia precyzyjne kontrolowanie uprawnień dostępowych. Z tego względu mechanizm ten jest powszechnie wykorzystywany w nowoczesnych aplikacjach internetowych, systemach opartych na API oraz integracjach z usługami zewnętrznymi.

6.4.4 JWT

JSON Web Token (JWT) jest standardem umożliwiającym bezpieczne przekazywanie informacji pomiędzy stronami w formie kompaktowego obiektu JSON. Token ten jest najczęściej wykorzystywany w systemach autoryzacji oraz uwierzytelniania, szczególnie w architekturze opartej o REST API i mikroserwisy. Standard JWT został opisany w specyfikacji RFC 7519 opracowanej przez IETF [58]. JWT składa się z trzech części oddzielonych kropkami:

- **Header (nagłówek)** - zawiera informacje o typie tokena oraz użytym algorytmie podpisu, np. HMAC SHA256 lub RSA,
- **Payload (ładunek)** - zawiera tzw. claims, czyli dane o użytkowniku i uprawnieniach, np. identyfikator użytkownika, rolę, zakres dostępu (scope) oraz czas ważności tokena,
- **Signature (podpis)** - służy do weryfikacji integralności tokena i zapewnia, że jego zawartość nie została zmodyfikowana.

Token JWT jest kodowany w formacie Base64¹⁰⁰, dzięki czemu może być łatwo przesyłany w nagłówkach http - najczęściej w polu *Authorization: Bearer <token>*.

Zasada działania

Po poprawnym uwierzytelnieniu użytkownika serwer generuje JWT i podpisuje go przy użyciu klucza prywatnego lub pary kluczy asymetrycznych. Następnie token jest zwracany do klienta, który przechowuje go lokalnie i dołącza do każdego żądania wysyłanego do API. Serwer zasobów weryfikuje podpis tokena i na jego podstawie decyduje o dostępie do zasobów.

Zalety JWT

- brak konieczności przechowywania sesji po stronie serwera,
- łatwa skalowalność systemów rozproszonych,
- szybka weryfikacja bez zapytań do bazy danych,
- możliwość przenoszenia danych o użytkowniku w tokenie.

Wady i zagrożenia

- brak możliwości łatwego unieważnienia tokena przed wygaśnięciem,
- ryzyko przechowywania zbyt dużej ilości danych w payload,
- konieczność bezpiecznego przechowywania tokena po stronie klienta,
- podatność na błędy implementacyjne (np. brak weryfikacji podpisu).

Z tego powodu JWT powinien być stosowany z zachowaniem zasad bezpieczeństwa, takich jak krótki czas życia tokena, użycie HTTPS oraz odpowiednie zarządzanie kluczami kryptograficznymi.

6.4.5 OWASP

OWASP, czyli (*Open Worldwide Application Security Project*) to organizacja zajmująca się poprawą bezpieczeństwa aplikacji internetowych. Jednym z jej najbardziej znanych projektów jest lista OWASP Top 10, która przedstawia najważniejsze i najczęściej występujące zagrożenia bezpieczeństwa aplikacji webowych, na które powinno zwrócić się szczególną uwagę implementując architekturę oraz systemy. Frameworki takie jak *Spring Boot* oraz zależność *Spring Security* automatycznie dba za nas o przestrzeganie reguł i zabezpieczenie przed zagrożeniami opisanymi w OWASP.

OWASP Top 10 (najczęstsze podatności)

- Broken Access Control - błędy w kontroli dostępu
- Cryptographic Failures - błędy w szyfrowaniu danych
- Injection - np. SQL Injection, Command Injection
- Insecure Design - błędy projektowe systemu
- Security Misconfiguration - niepoprawna konfiguracja systemu

¹⁰⁰ Schemat kodowania danych binarnych do postaci tekstowej, wykorzystywany m.in. w transmisji danych i systemach internetowych

- Vulnerable and Outdated Components - podatne biblioteki i zależności
- Identification and Authentication Failures - błędy logowania i sesji
- Software and Data Integrity Failures - brak kontroli integralności
- Security Logging and Monitoring Failures - brak monitorowania incydentów
- Server-Side Request Forgery (SSRF) - wymuszanie zapytań po stronie serwera

6.5. Zastosowana architektura bazy danych w systemie

W niniejszym podrozdziale przedstawiono architekturę bazy danych zastosowaną w implementowanym systemie. Projekt bazy danych stanowi jeden z kluczowych elementów systemu, ponieważ odpowiada za trwałe przechowywanie, organizację oraz integralność danych wykorzystywanych przez aplikację.

Przy projektowaniu struktury bazy danych uwzględniono wymagania funkcjonalne systemu oraz założenia dotyczące jego skalowalności i wydajności. Istotnym celem było zapewnienie spójności danych oraz optymalizacja operacji odczytu i zapisu, szczególnie w kontekście częstych zapytań wykonywanych przez warstwę aplikacyjną.

Zastosowana architektura została zaprojektowana w sposób umożliwiający łatwe rozszerzanie struktury danych w przyszłości, bez konieczności wprowadzania istotnych zmian w istniejącej logice systemu. W kolejnych częściach podrozdziału przedstawiono model logiczny bazy danych, zastosowane relacje pomiędzy encjami oraz przyjęte założenia projektowe. Systemowa baza danych została podzielona na 25 encji i jest współdzielona pomiędzy mikroserwisami mającymi do nich dostęp z zachowaniem odpowiedzialności, co oznacza że dany serwis korzysta tylko z tych encji, które są do niego przydzielone poprzez domenową logikę. Za zarządzanie, optymalizacje oraz migracje danych w systemie odpowiada narzędzie Liquibase, które upraszcza procesy tworzenia obiektów znajdujących się w bazie danych.

6.5.1 Relacyjne bazy danych

Relacyjne bazy danych stanowią model przechowywania danych, w którym informacje organizowane są w postaci tabel składających się z wierszy i kolumn. Każda tabela reprezentuje określoną encję, natomiast relacje pomiędzy tabelami są definiowane przy użyciu kluczy głównych i obcych.

Model relacyjny zapewnia wysoką spójność danych oraz umożliwia wykonywanie złożonych zapytań przy użyciu języka SQL. Dzięki zastosowaniu zasad normalizacji możliwe jest ograniczenie redundancji danych oraz poprawa ich integralności.

Relacyjne bazy danych są szeroko stosowane w systemach informatycznych, w których istotne jest zachowanie struktury danych oraz ich logicznych powiązań, co czyni je odpowiednim wyborem dla wielu aplikacji biznesowych.

6.5.2 Porównanie bazy danych współdzielonej, a oddzielnej dla każdej domeny lub mikroserwisu

W projektowaniu systemów informatycznych kluczową decyzją architektoniczną jest również sposób organizacji warstwy danych. Najczęściej rozważa się dwa podejścia:

wykorzystanie jednej wspólnej bazy danych dla całego systemu lub zastosowanie odrębnych baz danych przypisanych do poszczególnych domen bądź mikroservisów.

Model bazy współdzielonej

W podejściu tym wszystkie moduły systemu operują na jednej wspólnej bazie danych. Zapewnia to łatwiejszy dostęp do danych oraz upraszcza realizację spójności transakcyjnej pomiędzy modułami.

Jednocześnie takie rozwiązanie prowadzi do silnego powiązania komponentów systemu na poziomie danych, co może utrudniać dalszy rozwój aplikacji oraz ograniczać jej skalowalność. Wprowadzenie zmian w strukturze bazy często wymaga uwzględnienia wpływu na wiele części systemu.

Model baz rozdzielonych (mikroservis/domeny)

W tym podejściu każda domena lub mikroservis posiada własną, niezależną bazę danych. Komunikacja pomiędzy usługami odbywa się poprzez interfejsy API lub zdarzenia systemowe.

Rozwiązanie to zwiększa niezależność komponentów oraz umożliwia ich niezależne skalowanie i rozwój. Dodatkowo pozwala na dobór różnych technologii bazodanowych w zależności od potrzeb konkretnej domeny.

Jednakże wadą tego podejścia jest większa złożoność systemu, szczególnie w zakresie utrzymania spójności danych pomiędzy usługami oraz konieczność implementacji mechanizmów ACID na poziomie kilku baz danych. Model współdzielonej bazy danych lepiej sprawdza się w prostszych systemach monolitycznych, gdzie priorytetem jest spójność i prostota implementacji. Natomiast podejście z oddzielnymi bazami danych jest typowe dla architektury mikroservisowej, w której kluczowe znaczenie ma skalowalność, elastyczność oraz niezależność poszczególnych komponentów.

6.5.3 Uwzględnione optymalizacje bazy

W celu poprawy wydajności oraz efektywności działania bazy danych w systemie zastosowano kilka szczególnych optymalizacji. Ich głównym celem było skrócenie czasu wykonywania zapytań oraz zmniejszenie obciążenia warstwy bazodanowej.

Jednym z podstawowych zastosowanych rozwiązań było wykorzystanie indeksów na kluczowych kolumnach, które są często wykorzystywane w zapytaniach wyszukiwania oraz filtrowania danych. Pozwoliło to na znaczące przyspieszenie operacji odczytu. Zostały założone indexy pojedyncze jak i złożone na kilku kolumnach.

Dodatkowo ograniczono liczbę zbędnych zapytań poprzez optymalizację logiki po stronie aplikacji oraz stosowanie odpowiednich mechanizmów pobierania danych, takich jak selektywne zapytania zamiast pobierania pełnych zestawów danych, streamowanie danych w częściach oraz pobieranie czy zapisywanie danych jako zbiór bez robienia zbędnych pojedynczych operacji.

6.5.4 Indeksy

Indeksy są strukturami bazodanowymi pomagającymi w wyszukiwaniu danych na encjach. Dzięki nim możemy uzyskać szybszy dostęp do zasobów, natomiast jak przy tworzeniu architektury musimy też pamiętać, kiedy dany indeks jest w stanie nam pomóc, a kiedy staje się zbędny i utrudnia operacje na bazie na konkretnych rekordach, encjach oraz innych strukturach wykorzystywanych w bazie.

Stosowanie indeksów znacząco poprawia wydajność operacji odczytu, szczególnie w przypadku dużych zbiorów danych oraz często wykonywanych zapytań filtrujących i sortujących. Jednocześnie należy pamiętać, że indeksy mogą zwiększać czas operacji zapisu, ponieważ wymagają ich aktualizacji przy każdej modyfikacji danych.

6.5.5 Zwykle (pojedyncze)

Zwykle indeksy to najprostszy typ indeksów bazodanowych, tworzonych na jednej kolumnie tabeli. Ich głównym celem jest przyspieszenie operacji wyszukiwania, filtrowania oraz sortowania danych. W praktyce pozwalają one systemowi zarządzania bazą danych na uniknięcie pełnego skanowania tabeli, zastępując je bardziej efektywnymi strukturami dostępu do danych. W PostgreSQL indeksy tego typu najczęściej implementowane są za pomocą struktury B-Tree, która dobrze sprawdza się w większości typowych zastosowań. W naszym systemie indeksy na kolumny pojedyncze zakładane były szczególnie przy użyciu narzędzia Liquibase, który posiada możliwości tworzenia takich prostych struktur.

Listing 17: Przykład utworzenia indeksu pojedynczego na kolumnie `bank_account_id`

```
<createIndex tableName="landlord" indexName="ix_bank_account_id">
  <column name="bank_account_id"/>
</createIndex>
```

6.5.6 Złożone

Indeksy złożone są tworzone na więcej niż jednej kolumnie tabeli. Ich zastosowanie pozwala przyspieszyć wykonywanie zapytań, które filtrują lub sortują dane jednocześnie według kilku atrybutów. W odróżnieniu od indeksów pojedynczych uwzględniają one relację pomiędzy wartościami wielu kolumn, co może znacząco poprawić wydajność w bardziej złożonych operacjach.

W systemie *PostgreSQL* indeksy wielokolumnowe najczęściej wykorzystują strukturę B-tree. Podczas tworzenia takiego indeksu istotna jest kolejność kolumn, ponieważ wpływa ona bezpośrednio na sposób wykorzystania indeksu przez optymalizator zapytań.

Zastosowanie indeksów złożonych jest szczególnie korzystne w tabelach zawierających dużą liczbę rekordów, gdzie zapytania regularnie wykorzystują kilka kolumn jednocześnie - na przykład podczas wyszukiwania użytkowników według nazwiska i imienia.

6.5.7 Rozszerzenie indexów Btree GiSTs do bazy PostgreSQL

W systemie oprócz indeksów wspomnianych we wcześniejszych rozdziałach są utworzone indeksy specjalne. Ze względu na specjalistyczne operacje oraz główny wątek jakim są najmy, żeby móc bezproblemowo i spójnie procesować umowy musiał zostać utworzony indeks na kolumnach z datami. Uzyskanie odpowiednich możliwości związanych z blokowaniem nakładających się okresów umów podczas tworzenia najmów oraz spójność

czasowa wszystkich najmów została uzyskana dzięki rozszerzeniu *BTree GISTS* do bazy Postgres.

Listing 18: Implementacja rozszerzenia BTree GiSTs do bazy PostgreSQL

```
ALTER TABLE rental
  ADD COLUMN IF NOT EXISTS period daterange
  GENERATED ALWAYS AS (
    daterange(
      start_date,
      CASE WHEN is_indefinite THEN 'infinity'::date ELSE end_date END,
      '[' )
  )) STORED;
```

System PostgreSQL oferuje możliwość rozszerzania swojej funkcjonalności za pomocą dodatkowych modułów. Jednym z nich jest rozszerzenie *btree_gist*, które umożliwia wykorzystanie operatorów charakterystycznych dla klasycznych indeksów B-tree w ramach indeksów typu **GiST** (*Generalized Search Tree*). Rozwiązanie to łączy zalety dwóch mechanizmów indeksowania - prostotę i szerokie zastosowanie indeksów B-tree oraz większą elastyczność zapewnianą przez strukturę GiST.

6.6. DDD (Domain-Driven-Design)

Domain-Driven Design (DDD), czyli projektowanie sterowane domeną, to podejście do tworzenia oprogramowania, które koncentruje się na modelu biznesowym (domenie), a nie wyłącznie na aspektach technicznych. Takie podejście zmienia spojrzenie na dany system w kontekście konkretnych operacji biznesowych oraz ich podziału. Budując system oparty o ten model skupiamy się na domenach, czyli obszarach, w których będzie działa się hermetycznie zamknięta logika biznesowa dla poszczególnych struktur i ról w systemie. Możemy myśleć o tym jak o zamkniętych kontenerach, do których dostęp jest tylko przez wyznaczone ścieżki, a dane który znajdują się w nich nie mogą być jawnie i bezpośrednio dostępne dla innych kontenerów z oddzielną logiką biznesową.

DDD zostało spopularyzowane przez Erica Evansa w książce *“Domain-Driven Design: Tackling Complexity in the Heart of Software”*. Głównym celem tego podejścia jest lepsze odwzorowanie rzeczywistego świata biznesu w kodzie oraz skuteczniejsze radzenie sobie ze złożonymi systemami. Popularność na to podejście wzrosła ze względu na lepsze zrozumienie aspektów biznesowych poprzez osoby techniczne, dzięki czemu przekazywana wiedza biznesowa w łatwy sposób może zostać ujednolicona oraz umieszczona w jednym konkretnym obszarze systemu, który nie miesza się z innymi obszarami z inną wiedzą biznesową zaimplementowaną w systemie.

Czym jest domena?

Domena to obszar problemowy, który system ma obsługiwać i rozwiązywać. Może to być np.:

- Obszar odpowiedzialny za zarządzanie najmem (tworzenie, statusy, obsługa),

- Obszar odpowiedzialny za dokumenty (generowanie, zapisywanie),
- Moduł odpowiedzialny za zarządców (tworzenie, weryfikacja, dostęp).

DDD zakłada, że to **domena jest najważniejsza**, a nie frameworki, bazy danych czy interfejsy użytkownika.

6.6.1 Model domenowy

Model domenowy to logiczna reprezentacja problemu biznesowego. Składa się z takich elementów jak:

- **Encje (*Entities*)** - obiekty posiadające tożsamość oznaczające źródło prawdy w systemie. Są to dane które są trwałe poprzez ich zapis, prawdziwe oraz dostępne do operowania na nich tylko w przypisanej domenie, do której należą. Takim przykładem może być encja bazy danych mapowana na obiekt w systemie co pokazuje listing 19.

Listing 19: Mapowane encji bazodanowych na obiekty domenowe

```
@Entity
@Table(name = "landlord")
class LandlordEntity {
    @Builder.Default
    @OneToMany(
        mappedBy = "landlord",
        cascade = CascadeType.ALL,
        orphanRemoval = true,
        fetch = FetchType.LAZY
    )
    private Set<LandlordUserEntity> landlordUsers = new HashSet<>();
    @Column(name = "bank_account_id") private Long bankAccountId;
    @Column(name = "individual_id") private Long individualId;
    @Column(name = "company_id") private Long companyId;
    @Column(name = "address_id") private Long addressId;
}
```

- **Wartości (*Value Objects*)** - obiekty bez tożsamości, czyli dane wzięte z naszych encji domenowych i zmapowane na konkretne obiekty/interfejsy, które są dostępne pomiędzy domenami w momencie potrzebny ich w operacjach. Pobierane są przez określone ścieżki, które są zaimplementowane przez agregaty.

Listing 20: Przykład implementacji interfejsu z danymi

```
public interface Landlord {

    String getDisplayName();

    LandlordType getType();

    Long getId();
}
```

```

    String getPublicId();

    Status getStatus();

    String getEmail();

    BankAccount getBankAccount();

    String getPhoneNumber();

    Company getCompany();
}

```

- **Agregaty (*Aggregates*)** - grupy powiązanych obiektów traktowane jako całość. Są to zagnieżdżone serwisy innych domen potrzebne do wykonania określonych operacji w innej domenie biznesowej. Przykładem takiej logiki może być agregat do operacji na firmie, który jest powiązany z domeną zarządcy, gdyż zarządca może posiadać przypisaną firmę w systemie.

Listing 21: Utworzony agregat gotowy do zaimplementowania

```

public interface CompanyService {

    Long insert(@NonNull Company company);

    Optional<Company> get(long id);

    void update(long id, @NonNull Company company);
}

```

- **Serwisy domenowe (*Domain Services*)** - są to serwisy dostępne tylko w danej jednej domenie biznesowej, które operują na encjach, czyli obiektach tożsamości przypisanej domeny.

Listing 22: Implementacja serwisu domenowego

```

class LandlordServiceImpl implements LandlordService {
    private final LandlordRepository repository;
    private final AddressService addressService;
    private final IndividualService individualService;
    private final CompanyService companyService;
    private final BankAccountService bankAccountService;
    private final ApplicationEventPublisher eventPublisher;
    private final LandlordFactory factory;

    @Override
    @Transactional
    public Landlord insert(@NonNull Landlord landlord) {
        Log.info(

```

```

        "Inserting landlord domain for email: {}",
        landlord.getEmail()
    );
    Long addressId = addressService.insert(landlord.getAddress());

    LandlordEntity entity;
    if (LandlordType.COMPANY.equals(landlord.getType())) {
        Long companyId =
            companyService.insert(landlord.getCompany());
        entity = repository.save(buildEntity(
            landlord,
            addressId,
            companyId,
            null));
    }
    <...>
}
@Override
public Optional<Landlord> getBaseByPublicId(@NonNull String publicId) {
    LandlordEntity entity = repository.findByPublicId(publicId)
        .orElseThrow(() -> new IllegalStateException(format(
            "Not found landlord with public id %s",
            publicId)));
    BankAccount bankAccount = ofNullable(entity.getBankAccountId())
        .map(bankAccountId -> bankAccountService.get(bankAccountId)
        .orElseThrow(() -> new IllegalStateException(format(
            "Not found bank account for landlord with public id %s",
            publicId)
        )))
        .orElse(null);
    return Optional.of(factory.create(entity, bankAccount));
}
}

```

- **Repozytoria (*Repositories*)** - warstwa dostępu do danych, ograniczona do operacji domenowej, do której jest przypisana.

Listing 23: Implementacja repozytorium domenowego

```

@Repository
interface LandlordRepository extends JpaRepository<LandlordEntity, Long> {

    @Query("SELECT l.id FROM LandlordEntity l WHERE l.publicId = :publicId")
    Optional<Long> findByIdByPublicId(@Param("publicId") String publicId);

    List<LandlordEntity> findByIdIn(Collection<Long> ids);

    Optional<LandlordEntity> findByPublicId(String publicId);
}

```

6.6.2 Bounded Context (Ograniczony kontekst)

W dużych systemach różne części mogą używać tych samych słów, ale znaczących coś innego. Np.:

- „Klient” w systemie sprzedaży,
- „Klient” w systemie wsparcia technicznego.

DDD rozwiązuje ten problem przez (*bounded contexts*), czyli wydzielone obszary, w których obowiązuje jeden spójny model.

6.6.3 Wspólny język biznesowy

Jednym z kluczowych elementów DDD jest wspólny język, używany przez:

- programistów,
- analityków,
- ekspertów biznesowych.

Oznacza to, że wszyscy używają tych samych pojęć zarówno w rozmowach, jak i w kodzie. Jeśli w biznesie używa się terminu „najem”, to w kodzie również powinno istnieć Rental, a nie jako ObjectData czy TransactionRecord.

6.6.4 Warstwy architektury DDD

Typowa architektura DDD składa się z warstw:

- **Warstwa prezentacji** - interfejs użytkownika / API,
- **Warstwa aplikacji** - orkiestracja przypadków użycia,
- **Warstwa domeny** - logika biznesowa (serce systemu),
- **Warstwa infrastruktury** - baza danych, integracje, API zewnętrzne.

Najważniejsza jest warstwa domeny - powinna być niezależna od technologii.

6.6.5 Zalety rozwiązania

Lepsze odwzorowanie rzeczywistego biznesu w kodzie

Największą zaletą DDD jest to, że kod zaczyna być językiem biznesu. Zamiast technicznych struktur, które trzeba tłumaczyć, w systemie pojawiają się pojęcia znane z domeny, takie jak „Najem”, „Płatność” czy „Rezerwacja”. Dzięki temu:

- łatwiej zrozumieć, co system robi,
- łatwiej wprowadzać zmiany biznesowe,
- mniej jest nieporozumień między zespołem, a klientem

Lepsza organizacja złożonych systemów

DDD szczególnie dobrze działa w dużych systemach, gdzie logika biznesowa jest skomplikowana. Dzięki pojęciom takim jak:

- ograniczone konteksty (*bounded contexts*)
- agregaty
- serwisy domenowe

system można podzielić na mniejsze, logiczne części, które są łatwiejsze do utrzymania i rozwijania.

Zmiany biznesowe są łatwiejsze do wprowadzenia

Ponieważ logika biznesowa znajduje się w jednym miejscu (warstwie domenowej), a nie jest rozproszona po kontrolerach, bazie danych i UI, zmiany są bardziej lokalne. To oznacza:

- mniej efektów ubocznych,
- mniejsze ryzyko „zepsucia” innych części systemu,
- szybsze reagowanie na zmiany w decyzjach projektowych

6.6.6 Wady rozwiązania

Duża złożoność wejściowa

DDD nie jest prostym podejściem na początek, a dla małych projektów może to być przerost formy nad treścią. Wymaga zrozumienia wielu pojęć, takich jak:

- agregaty,
- ograniczone konteksty (*bounded context*),
- encje i (*value objects*),
- serwisy domenowe.

Wysoki próg współpracy z biznesem

DDD zakłada stałą współpracę z ekspertami domenowymi, a bez dobrej komunikacji ten model traci sens. W praktyce bywa to trudne, ponieważ:

- biznes nie zawsze ma czas,
- wymagania są niejasne lub zmienne,
- komunikacja nie zawsze jest płynna.

Ryzyko nadmiernego skomplikowania systemu

Jednym z częstych problemów jest próba „na siłę” stosowania DDD tam, gdzie nie jest potrzebne. Efektem później są:

- zbyt wiele klas i warstw,
- trudny do zrozumienia kod,
- wolniejszy rozwój prostych funkcji.

DDD powinno być stosowane tam, gdzie złożoność biznesowa tego wymaga, a nie w każdym miejscu.

Większy nakład pracy na początku projektu

Modelowanie domeny, definiowanie kontekstów i budowanie poprawnej architektury wymagają czasu, a zysk pojawia się dopiero po określonym czasie. Na starcie:

- rozwój funkcjonalności może być wolniejszy,
- więcej czasu poświęca się na analizę niż napisanie oraz implementację kodu.

Potrzebny jest doświadczony zespół inżynierów

DDD nie jest podejściem dla początkujących osób w pełnym zakresie, ponieważ bez tego łatwo wpaść w błędne interpretacje. Zespół musi:

- rozumieć architekturę systemów,
- umieć modelować domenę,
- podejmować dobre decyzje projektowe.

6.7. Zastosowane wzorce projektowe

W procesie projektowania i implementacji systemu istotną rolę odegrały wzorce projektowe, które stanowią sprawdzone i wielokrotnie przetestowane rozwiązania typowych problemów występujących w inżynierii oprogramowania. Ich zastosowanie pozwala nie tylko na uporządkowanie struktury kodu, ale również na zwiększenie jego czytelności, elastyczności oraz łatwości dalszego utrzymania.

Głównym celem wykorzystania wzorców projektowych w tym systemie było zapewnienie wysokiej spójności architektury oraz wyraźnego rozdzielenia odpowiedzialności pomiędzy poszczególnymi komponentami aplikacji. Dzięki temu logika biznesowa została oddzielona od warstw odpowiedzialnych za komunikację z użytkownikiem oraz dostęp do danych, co znacząco ułatwia rozwój i testowanie systemu.

Zastosowane wzorce projektowe są ściśle powiązane z podejściem **Domain-Driven Design**, które stanowi fundament architektury systemu. Wzorce w następnych podrozdziałach takie jak Factory czy Strategia wspierają modelowanie domeny i umożliwiają enkapsulację logiki biznesowej w sposób zgodny z rzeczywistymi procesami zachodzącymi w analizowanej domenie. Z kolei wzorce architektoniczne, takie jak *Dependency Injection* czy MVC, wspomagają budowę modularnej i skalowalnej struktury aplikacji.

6.7.1 Builder

Wzorzec **Builder** jest kreacyjnym wzorcem projektowym, którego celem jest oddzielenie procesu tworzenia złożonego obiektu od jego reprezentacji. Pozwala on na konstruowanie obiektów krok po kroku, umożliwiając większą kontrolę nad procesem ich powstawania.

Zastosowanie wzorca Builder jest szczególnie przydatne w sytuacjach, gdy obiekt posiada wiele opcjonalnych parametrów lub jego poprawna inicjalizacja wymaga wykonania kilku etapów. Dzięki temu kod staje się bardziej czytelny, a proces tworzenia obiektów bardziej przejrzysty i mniej podatny na błędy.

W kontekście systemu wzorzec Builder wspiera utrzymanie spójności modeli domenowych oraz poprawia ergonomię tworzenia obiektów w warstwie aplikacyjnej i

domenowej. Zastosowanie Lombokowej adnotacji *@Builder* tworzy z obiekt dostosowany to implementacji tego wzorca co pokazuje listing 24.

Listing 24: Adnotacja *@Builder* tworzy obiekt przygotowany pod ten wzorec, który możemy użyć za pomocą statycznej metody *builder()*

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@JsonInclude(JsonInclude.Include.NON_NULL)
public class LandlordDTO {
    private String displayName;
    private String type;
    private String status;
    private String publicId;
    private BankAccountDTO bankAccount;
    private String email;
    private String phoneNumber;
}

LandlordDTO.builder()
    .displayName(landlord.getDisplayName())
    .type(landlord.getType())
    .status(landlord.getStatus())
    .publicId(landlord.getPublicId())
    .bankAccount(ofNullable(landlord.getBankAccount())
        .map(this::toDto)
        .orElse(null))
    .email(landlord.getEmail())
    .phoneNumber(landlord.getPhoneNumber())
    .build();
```

6.7.2 ACL

Wzorec **ACL** (*Anti-Corruption Layer*) jest elementem architektonicznym stosowanym w systemach opartych na podejściu Domain-Driven Design. Jego głównym celem jest ochrona modelu domenowego przed „zanieczyszczeniem” wynikającym z integracji z zewnętrznymi systemami, które mogą posługiwać się odmiennym modelem danych lub logiką biznesową.

Warstwa ACL pełni rolę pośrednika pomiędzy systemem, a zewnętrznymi zależnościami. Odpowiada za tłumaczenie struktur danych oraz mapowanie operacji z zewnętrznego świata na model obowiązujący wewnątrz systemu. Dzięki temu model domenowy pozostaje spójny i niezależny od decyzji projektowych podjętych poza systemem.

Zastosowanie ACL zwiększa izolację warstwy domenowej, ułatwia integrację z różnymi systemami zewnętrznymi oraz zmniejsza ryzyko przenikania niepożądanych zależności do rdzenia aplikacji.

Listing 25: Implementacja wzorca ACL podczas mapowania danych wejściowych na domenowe

```
record RegisterLandlordAdapter(  
    @NonNull RegisterLandlordDTO dto  
) implements RegisterLandlord {  
  
    @Override  
    public String getDisplayName() {  
        return dto.getDisplayName();  
    }  
  
    @Override  
    public String getType() {  
        return dto.getType();  
    }  
  
    @Override  
    public String getEmail() {  
        return dto.getEmail();  
    }  
  
    @Override  
    public String getPhoneNumber() {  
        return dto.getPhoneNumber();  
    }  
  
    @Override  
    public Company getCompany() {  
        return ofNullable(dto.getCompany())  
            .map(CompanyAdapter::new)  
            .orElse(null);  
    }  
}
```

6.7.3 Adapter

Wzorzec **Adapter** jest strukturalnym wzorcem projektowym, którego głównym celem jest umożliwienie współpracy klas posiadających niekompatybilne interfejsy. Realizuje to poprzez „opakowanie” istniejącej klasy w nową warstwę, która dostosowuje jej sposób działania do oczekiwań pozostałych elementów systemu. W praktyce wzorzec ten wspiera utrzymanie przejrzystej architektury oraz zwiększa elastyczność systemu, umożliwiając łatwiejszą wymianę lub rozbudowę zewnętrznych zależności. Ten wzorzec wygląda podobnie do wzorca ACL natomiast jego rolą jest zmiana struktury oraz zakrycie potrzebnych implementacji potrzebnych do operacji pomiędzy domenami wewnątrz pod interfejsem.

Listing 26: Implementacja wzorca Adapter

```
private record CompanyAdapter(@NonNull CompanyDTO dto) implements Company {  
  
    @Override  
    public String getName() {  
        return dto.getName();  
    }  
}
```

```

    }

    @Override
    public String getTaxIdentifier() {
        return dto.getTaxIdentifier();
    }

    @Override
    public String getRegon() {
        return dto.getRegon();
    }

    @Override
    public Boolean getActive() {
        return dto.getActive();
    }

    @Override
    public CompanyType getType() {
        return ofNullable(dto.getType())
            .map(CompanyType::valueOf)
            .orElse(null);
    }
}

```

6.7.4 Strategia

Wzorzec **Strategia** jest behawioralnym wzorcem projektowym, którego celem jest definiowanie rodziny algorytmów, enkapsulowanie¹⁰¹ ich i umożliwienie ich wzajemnej wymiany w trakcie działania systemu. Pozwala to na wybór konkretnego wariantu logiki w zależności od kontekstu, bez konieczności modyfikowania kodu klienta.

Dzięki zastosowaniu wzorca Strategia możliwe jest oddzielenie logiki decyzyjnej od implementacji poszczególnych algorytmów. W efekcie system staje się bardziej elastyczny, czytelny oraz łatwiejszy w rozbudowie, ponieważ nowe warianty zachowania można dodawać bez ingerencji w istniejący kod. W kontekście systemu wzorzec Strategia umożliwia dynamiczne dostosowanie sposobu realizacji wybranych operacji biznesowych, co zwiększa jego elastyczność oraz ułatwia utrzymanie spójnej architektury.

Listing 27: Utworzenie klasy abstrakcyjnej *DocumentBuilder* z przekazaniem typu generycznego do implementacji wzorca dla obiektów dziedziczących po niej

```

public abstract class DocumentBuilder<T> {

    public abstract DocumentMessageDTO<T> build(@NonNull Document document);

    protected String buildLandlordIdentifier(@NonNull Landlord landlord) {
        if (LandlordType.COMPANY.equals(landlord.getType())) {
            return landlord.getCompany().getTaxIdentifier();
        }
    }
}

```

¹⁰¹ Jedną z podstawowych zasad programowania obiektowego, polegającą na ukrywaniu wewnętrznej implementacji obiektu i udostępnianiu dostępu do jego danych wyłącznie poprzez zdefiniowany interfejs (np. metody publiczne)

```

    } else if (LandlordType.INDIVIDUAL.equals(landlord.getType())) {
        return landlord.getIndividual().getNationalIdentifierNumber();
    }

    throw new IllegalStateException(
        "Not found landlord type - " + landlord.getType());
}

// Użycie metody build() niezależnie od implementacji wewnątrz
@Override
public void send(@NonNull Document document) {
    client.send(resolveTemplateModel(document.getType()).build(document));
}

```

6.7.5 Fabryka

Wzorzec **Fabryka** (*Factory*) jest kreacyjnym wzorcem projektowym, którego celem jest oddzielenie procesu tworzenia obiektów od miejsca ich wykorzystania. Dzięki temu logika odpowiedzialna za inicjalizację i konfigurację obiektów zostaje skupiona w dedykowanym komponencie, co zwiększa przejrzystość oraz spójność kodu.

Zastosowanie wzorca Fabryka pozwala ukryć szczegóły związane z tworzeniem obiektów oraz ograniczyć zależność pomiędzy klasami korzystającymi z tych obiektów, a ich konkretnymi implementacjami. Takie podejście ułatwia rozwój systemu, ponieważ zmiany w sposobie tworzenia obiektów nie wymagają modyfikacji kodu korzystającego z nich. W kontekście systemu wzorzec Fabryka wspiera utrzymanie zasad enkapsulacji oraz pozwala na centralizację logiki odpowiedzialnej za tworzenie elementów domenowych, co pozytywnie wpływa na czytelność, testowalność i łatwość dalszej rozbudowy aplikacji.

Listing 28: Implementacja wzorca Fabryki, aby móc dostosować wychodzące obiekty z domeny w odpowiedni sposób do odpowiednich operacji kontekstowych

```

class LandlordFactory {
    Landlord create(@NonNull LandlordEntity entity) {
        return LandlordImpl.builder()
            .id(entity.getId())
            .displayName(entity.getDisplayName())
            .type(entity.getType())
            .publicId(entity.getPublicId())
            .status(entity.getStatus())
            .email(entity.getEmail())
            .phoneNumber(entity.getPhoneNumber())
            .build();
    }

    @Getter
    @Builder
    private static class LandlordImpl implements Landlord {
        private final String displayName;
        private final LandlordType type;
        private final String publicId;
        private final Status status;
    }
}

```

```

        private final Long id;
        private final String email;
        private final BankAccount bankAccount;
        private final String phoneNumber;
        private final Company company;
        private final Address address;
        private final Individual individual;
    }
}

```

6.8. Metodologia SOLID

SOLID to zbiór pięciu zasad projektowania obiektowego, których celem jest tworzenie oprogramowania łatwego w utrzymaniu, rozwijaniu i testowaniu. Zasady te zostały spopularyzowane przez Robert C. Martin i obecnie są uznawane za jedno z podstawowych wytycznych stosowanych podczas projektowania systemów informatycznych. Stosowanie zasad SOLID pozwala ograniczyć złożoność kodu, zmniejszyć liczbę błędów oraz ułatwić wprowadzanie zmian w przyszłości.

W praktyce rozwój oprogramowania wiąże się z ciągłymi zmianami wymagań biznesowych. Kod napisany bez odpowiednich zasad projektowych często staje się trudny do zrozumienia i rozbudowy. W takich sytuacjach nawet niewielka modyfikacja może prowadzić do nieoczekiwanych błędów w innych częściach systemu. Zasady SOLID stanowią odpowiedź na ten problem, ponieważ promują tworzenie modułów o jasno określonych odpowiedzialnościach oraz ograniczają nadmierne powiązania między elementami aplikacji.

System został zbudowany również w oparciu o te zasady, natomiast nie wszystkie implementacje są dopasowane w całości. Ze względu na architekturę opartą o DDD pewne schematy nałożone były z góry, co utrudniało stosowanie się do zasad całości SOLID.

6.8.1 Single Responsibility Principle (SRP)

Pierwsza zasada mówi, że każda klasa powinna mieć tylko jedną odpowiedzialność. Oznacza to, że powinna istnieć tylko jedna przyczyna, dla której dana klasa wymaga zmiany. Jeśli jeden element systemu odpowiada za wiele różnych zadań, staje się trudniejszy do zrozumienia i utrzymania. Rozdzielenie obowiązków pomiędzy mniejsze komponenty zwiększa przejrzystość kodu oraz ułatwia jego rozwój. Ta zasada była stosowana w systemie od początku kreowania architektury ze względu na charakterystykę DDD, która sama w sobie kieruje architekturę do rozdzielania odpowiedzialności na poszczególne domeny, poddomeny i serwisy położone wewnątrz.

6.8.2 Open/Closed Principle (OCP)

Zasada otwarte-zamknięte zakłada, że oprogramowanie powinno być otwarte na rozszerzanie, ale zamknięte na modyfikację. W praktyce oznacza to, że nowe funkcjonalności powinny być dodawane bez konieczności zmiany istniejącego kodu. Takie podejście zmniejsza ryzyko wprowadzenia błędów do już działających części systemu i pozwala łatwiej dostosowywać aplikację do nowych wymagań. System stosuje wiele możliwości tej zasady, jedną z nich jest Bounded Context, dzięki któremu za pomocą interesuj może implementować klasy kontekstowe w zależności od tego co w danych operacjach pomiędzy domenami

potrzebujemy. Przykładem jest interfejs *Landlord*, który może być zaimplementowany w różnych kontekstach, taki interfejs ukazany jest na listingu 29.

Listing 29: Przykład interfejsu Landlord oraz możliwych implementacji

```
record RegisterLandlordAdapter(  
    @NonNull RegisterLandlordDTO dto  
) implements Landlord {  
  
    @Override  
    public String getDisplayName() {  
        return dto.getDisplayName();  
    }  
  
    @Override  
    public String getType() {  
        return dto.getType();  
    }  
  
    @Override  
    public String getEmail() {  
        return dto.getEmail();  
    }  
  
    @Override  
    public String getPhoneNumber() {  
        return dto.getPhoneNumber();  
    }  
  
    @Override  
    public Company getCompany() {  
        return ofNullable(dto.getCompany())  
            .map(CompanyAdapter::new)  
            .orElse(null);  
    }  
}  
  
// W zależności od kontekstu implementujemy różne rekordy z tych samych  
interfejsów  
record LandlordRegister implements Landlord {...}  
record Landlord implements Landlord {...}
```

6.8.3 Liskov Substitution Principle (LSP)

Zasada podstawienia Liskov odnosi się do dziedziczenia. Mówi ona, że obiekty klas pochodnych powinny móc zastępować obiekty klas bazowych bez wpływu na poprawność działania programu. Dzięki temu hierarchia klas pozostaje logiczna i spójna, a kod korzystający z klas bazowych nie wymaga dodatkowych modyfikacji podczas używania klas dziedziczących.

6.8.4 Interface Segregation Principle (ISP)

Zgodnie z zasadą segregacji interfejsów klasy nie powinny być zmuszane do implementowania metod, których nie wykorzystują. Zamiast tworzenia dużych i rozbudowanych interfejsów zaleca się projektowanie mniejszych, wyspecjalizowanych interfejsów. Pozwala to ograniczyć zbędne zależności i zwiększa czytelność całego projektu. Jednym z przykładów może posłużyć nam interfejs *BankAccountService*, którego użycie na serwisie będzie potrzebować implementacji tych dwóch metod.

Listing 30: Przykład interfejsu gotowego do implementacji na klasie

```
public interface BankAccountService {  
  
    Optional<Long> upsert(Long id, @NonNull BankAccount bankAccount);  
  
    Optional<BankAccount> get(long id);  
  
}
```

6.8.5 Dependency Inversion Principle (DIP)

Ostatnia zasada wskazuje, że moduły wysokiego poziomu nie powinny zależeć bezpośrednio od modułów niskiego poziomu. Obie grupy powinny opierać się na abstrakcjach, takich jak interfejsy lub klasy abstrakcyjne. Dzięki temu poszczególne elementy systemu są mniej zależne od konkretnych implementacji, co ułatwia testowanie oraz rozwijanie aplikacji. Zastosowanie tej zasady polega na tym, że zamiast odwoływać się do konkretnych klas, korzystamy z interfejsów, które następnie są implementowane przez klasy lub serwisy zależne od danego kontekstu. Takie podejście zwiększa elastyczność systemu i pozwala na łatwą podmianę implementacji bez konieczności modyfikowania kodu wyższego poziomu.

SOLID jest zbiorem zasad wspierających tworzenie przejrzystego, elastycznego i łatwego w utrzymaniu oprogramowania. Stosowanie tych reguł pozwala ograniczyć liczbę błędów, ułatwia rozwój systemów informatycznych oraz zwiększa ich odporność na zmiany. Z tego względu zasady SOLID są powszechnie wykorzystywane podczas projektowania współczesnych aplikacji i stanowią ważny element dobrych praktyk programistycznych

6.9. Implementacja aplikacji webowych - frontend

Warstwa frontendowa systemu zarządzania najmem nieruchomości składa się z dwóch osobnych aplikacji webowych: **landlord-frontend**, przeznaczonej dla wynajmujących, oraz **tenant-frontend**, przeznaczonej dla najemców. Obie aplikacje zostały zbudowane w oparciu o framework Angular 19 z wykorzystaniem języka TypeScript.

Podział na dwie niezależne aplikacje wynika z odmiennych wymagań funkcjonalnych każdej z grup użytkowników. Wynajmujący zarządza nieruchomościami, umowami i płatnościami, natomiast najemca przegląda dostępne oferty i śledzi stan swojego najmu. Rozdzielenie aplikacji pozwala na niezależne wdrożenia oraz precyzyjną kontrolę dostępu. Każda aplikacja komunikuje się wyłącznie z dedykowanym dla niej serwisem backendowym.

6.9.1 Prywatne współdzielona biblioteka npm dla aplikacji frontendowych

Dzisiejsze systemy informatyczne coraz częściej budowane są jako niezależne aplikacje zamiast jednego monolitu. Takie podejście rodzi pytanie: jak współdzielić kod między oddzielnymi projektami bez powielania?

Problem ten szczególnie jest widoczny w aplikacjach frontendowych. Aplikacje przeznaczone dla różnych grup użytkowników operują na tych samych modelach danych, podlegają tym samym regułom walidacji i prezentują spójny interfejs. Implementowanie tych elementów osobno w każdej aplikacji prowadzi do sytuacji, w której poprawki błędów wprowadzane są w wielu miejscach, a kod z czasem zaczyna się różnić między projektami.

Rozwiązaniem jest wydzielenie wspólnego kodu do osobnej, wersjonowanej biblioteki dystrybuowanej przez menadżera pakietów npm. Ponieważ biblioteka zawiera wewnętrzną logikę biznesową, użycie publicznego rejestru *npmjs.com*¹⁰² jest wykluczone. Zamiast tego stosuje się prywatny rejestr oferowany przez platformę *GitLab*.

Kontekst i motywacja

Warstwa frontendowa aplikacji do najmów składa się z dwóch niezależnych aplikacji frontendowych: **landlord-frontend** (panel wynajmującego) oraz **tenant-frontend** (panel najemcy). Obie aplikacje zostały zbudowane w technologii Angular 19 i współdzielą wiele elementów. Elementami używanymi wspólnie są: komponenty UI, modele domenowe, walidatory formularzy oraz serwisy pomocnicze. Bez mechanizmu współdzielenia kodu każda zmiana (np. nowa reguła walidacji PESEL-u) musiałaby być powielona w dwóch miejscach, co prowadzi do braku spójności w implementacji i podatności na błędy.

Struktura biblioteki

Biblioteka zbudowana jest jako projekt Angular Library z jednym punktem konfiguracyjnym, którym jest plik *public-api.ts*¹⁰³, eksportujący zawartość czterech pakietów: **modules**, **models**, **utils**, **validators** oraz **services**.

¹⁰² Oficjalna strona i rejestr pakietów dla ekosystemu Node.js (npm), służący do publikowania, wyszukiwania i instalowania bibliotek JavaScript

¹⁰³ Plik konfiguracyjny angularowej biblioteki pozwalający umieszczać pakiety i pliki dostępne publicznie

-
- ▼  projects
 - ▼  frontend-core
 - ▼  src
 - ▼  lib
 - ▼  models
 - >  document
 - >  landlord
 - >  notification
 - >  payment
 - >  property
 - >  range
 - >  realm
 - >  rental
 - >  tenant
 -  index.ts
 - ▼  modules
 - >  input-base
 - >  input-checkbox
 - >  input-datepicker
 - >  input-number
 - >  input-otp
 - >  input-select
 - >  input-text
 - >  loaders
 - >  notification
 - >  textarea-input
 -  abstract-input.component.ts
 -  extended-mask.config.ts
 -  index.ts
 - >  services
 - >  utils
 - >  validators
 - >  styles
 -  public-api.ts
-

Pakiet modules zawiera wielokrotnego użytku komponenty Angular m.in. bazowy komponent tekstowy (*InputTextComponent*), komponent numeryczny (*NumberInputComponent*), komponent wyboru z listy (*SelectComponent*), pole tekstowe wieloliniowe (*TextareaInputComponent*), a także dwa komponenty ładowania (*LoaderComponent*, *GlobalLoaderComponent*) i komponent powiadomień (*NotificationComponent*). Wszystkie komponenty nadrzędne dziedziczą po abstrakcyjnej klasie *AbstractInputComponent* przyjmującej generyczny typ wejściowy i wyjściowy, która implementuje interfejs Angular *ControlValueAccessor*. Dzięki temu są one w pełni kompatybilne z Angular Reactive Forms.

Pakiet models zawiera współdzielone typy domenowe: stałe opisujące status i typ nieruchomości (*PropertyStatus*, *PropertyType*), typ wynajmującego (*LandlordType*) oraz typ powiadomienia (*NotificationType*).

Pakiet utils grupuje funkcje pomocnicze: walidatory formularzy (*FormValidatorUtils*) z logiką weryfikacji hasła, NIP-u i PESEL-u, a także narzędzia do formatowania i parsowania numerów telefonu, kodu pocztowego i danych wejściowych pól numerycznych.

Pakiet services zawiera *NotificationService* - serwis zarządzający wyświetlaniem powiadomień w aplikacji. Poza kodem TypeScript biblioteka eksportuje również pliki stylów SCSS, czyli zmienne globalne (*_variables.scss*) oraz style wspólnych komponentów (*_buttons.scss*), dostępne do importu w aplikacjach konsumenckich.

Oprócz tego znajduje się tam wiele innych struktur, modeli i komponentów, które są współdzielone pomiędzy dwoma frontendami. W przypadku dodania nowej domeny i potrzeby zbudowania nowego projektu, wiele z komponentów jest już gotowych.

AbstractInputComponent - wzorzec ControlValueAccessor

Podstawą komponentów wejściowych dla formularzy jest abstrakcyjna klasa generyczna *AbstractInputComponent*, implementująca interfejs Angular *ControlValueAccessor*. Dzięki temu wszystkie komponenty input¹⁰⁴ takie jak (*InputTextComponent*, *NumberInputComponent*, *SelectComponent*, *TextareaInputComponent*) są w pełni kompatybilne z Angular Reactive Forms i mogą być używane wewnątrz FormGroup identycznie jak natywne pola html `<input>`.

Listing 31: Abstrakcyjna klasa *AbstractInputComponent* dostępna do dziedziczenia przez współdzieloną pola formularzy

```
export abstract class AbstractInputComponent<T> implements ControlValueAccessor {
  protected _value: T | null = null;
  // ...
}
```

NotificationService - stos powiadomień z Angular CDK Overlay

Serwis *NotificationService* zarządza stosem powiadomień wyświetlanych w prawym górnym rogu aplikacji. Korzysta z *Angular CDK Overlay*, co eliminuje potrzebę ręcznego

¹⁰⁴ Komponent technologii HTML stosowany jako element formularza `<input>`

zarządzania DOM-em¹⁰⁵. Każde powiadomienie jest dynamicznie renderowane jako *ComponentPortal*, a jego pozycja jest automatycznie przeliczana po zamknięciu sąsiedniego powiadomienia.

Listing 32: Metody pozwalające na wyświetlenie powiadomień o konkretnym typie

```
this.show(message, 'success'); // green toast
this.show(message, 'error');   // red toast
```

Walidatory domenowe

Klasa *FormValidatorUtils* zawiera walidatory Angular specyficzne dla polskiego prawa i kontekstu biznesowego.

- **passwordValidator** - reguły złożoności hasła (duże/małe litery, cyfra, znak specjalny, 8-64 znaki),
- **taxIdentifierValidator** - weryfikacja NIP (algorytm sum kontrolnych z wagami),
- **nationalIdentifierNumberValidator** - weryfikacja PESEL (suma kontrolna wg. wag [1,3,7, 9...]),
- **nationalIdentifierNumberIsAdultValidator** - sprawdzenie czy PESEL wskazuje osobę pełnoletnią,
- dodatkowo metody pozwalające na wyliczenie wieku.

6.9.2 The Angular Component Development Kit

Angular Component Development Kit, w skrócie CDK, jest oficjalną biblioteką dostarczaną przez zespół Angular jako część pakietu *@angular/cdk*. Jej zadaniem jest udostępnienie programistom zestawu gotowych zachowań i mechanizmów interfejsu użytkownika, które nie są powiązane z żadnym konkretnym wyglądem wizualnym. CDK stanowi warstwę pośrednią między surowym kodem Angular, a gotowymi bibliotekami komponentów, takimi jak *Angular Material*. Zapewnia logikę, pozostawiając pełną swobodę w kwestii stylowania, działania i optymalizacji komponentów.

Biblioteka CDK dostarcza moduły obsługujące między innymi nakładki na ekranie, czyli (*Overlay*), mechanizm portali, obsługuje przeciąganie i upuszczanie, zarządzanie focusem¹⁰⁶. Dodatkowo daje możliwość przewijania dużych list oraz wiele innych narzędzi. Dzięki temu programista nie musi implementować tych mechanizmów od podstaw, zamiast tego opiera się na sprawdzonym rozwiązaniu.

Zastosowanie w projekcie

¹⁰⁵ Document Object Model - model obiektowy dokumentu, który reprezentuje strukturę strony HTML jako drzewo elementów. Dzięki DOM język JavaScript może dynamicznie zmieniać treść, styl i strukturę strony w przeglądarce

¹⁰⁶ Stan elementu strony, który jest aktualnie aktywny i może odbierać dane wejściowe (np. wpisywanie tekstu w polu formularza po kliknięciu). W JavaScript można ten stan kontrolować i obsługiwać za pomocą zdarzeń

W projekcie CDK zostało wykorzystane do implementacji systemu powiadomień w prywatnej bibliotece frontend-core. Użyto dwóch elementów takich jak *Overlay* oraz mechanizmu *ComponentPortal*.

Overlay to mechanizm tworzenia elementów renderowanych ponad resztą interfejsu, niezależnie od struktury drzewa DOM. Każde powiadomienie systemowe jest tworzone jako osobna nakładka z precyzyjnie zdefiniowaną pozycją na ekranie. Znajduje się w prawym górnym rogu, z dynamicznie obliczanym przesunięciem pionowym zależnym od liczby aktualnie widocznych już powiadomień.

Listing 33: Implementacja mechanizmu Overlay

```
const overlayRef = this.overlay.create({
  hasBackdrop: false,
  positionStrategy: this.overlay
    .position()
    .global()
    .top(`${this.calculateOffset()}px`)
    .right('20px'),
});
```

ComponentPortal pozwala na dynamiczne renderowanie komponentu Angular wewnątrz stworzonej nakładki. Zamiast wstawiać powiadomienie do istniejącego drzewa komponentów aplikacji, serwis tworzy instancję *NotificationComponent* w trakcie i przyczepia ją do nakładki, przekazując dane wejściowe bezpośrednio przez referencję do instancji.

Listing 34: Implementacja mechanizmu ComponentPortal

```
const portal = new ComponentPortal(NotificationComponent, null, this.injector);
const componentRef = overlayRef.attach(portal);
componentRef.instance.message = message;
componentRef.instance.type = type;
componentRef.instance.onClose = () => this.remove(overlayRef)
```

Zastosowanie CDK Overlay w porównaniu do klasycznego podejścia opartego na warunkowym renderowaniu elementów w szablonie daje kilka korzyści. Powiadomienia istnieją poza drzewem komponentów aplikacji, co eliminuje ryzyko konfliktów z właściwościami CSS na elementach nadrzędnych. Każde powiadomienie ma własny cykl życia i własną nakładkę, co upraszcza zarządzanie stanem. Pozycjonowanie odbywa się po stronie CDK, a nie przez ręczne obliczenia przy użyciu CSS, co czyni kod bardziej czytelny i odpornym na zmiany layoutu.

6.9.3 The Angular Material

Angular Material jest oficjalną biblioteką komponentów UI dla frameworka Angular, tworzoną i utrzymywaną przez zespół Google. Dostarcza gotowe, w pełni dostępne komponenty wizualne oparte na specyfikacji *Material Design*. Przykładami są przyciski, pola

formularzy, tabele, okna dialogowe, selektory daty i wiele innych. Komponenty Angular Material są ściśle zintegrowane z ekosystemem Angular. Obsługują one Angular Reactive Forms, CDK, animacje i mechanizmy wstrzykiwania zależności.

Zastosowanie w projekcie

W projekcie Angular Material został użyty jedynie dla komponentu wyboru daty (*InputDatepickerComponent*) wbudowanego w bibliotekę frontend-core. Decyzja ta wynika z faktu, że samodzielna implementacja kalendarza byłaby bardziej czasochłonna do zaimplementowania niż użycie sprawdzonego rozwiązania. Wynika to z tego, że gotowe rozwiązania posiada obsługę zakresów dat, blokowania przedziałów, dostępności i formatowania.

Komponent *InputDatepickerComponent* rozszerza *AbstractInputComponent* z typem generycznym *string* i integruje dwie dyrektywy Angular Material: *MatDatepicker* - moduł kalendarza oraz *MatDatepickerInput* - dyrektywa wiążąca pole input z kalendarzem. Dzięki temu kliknięcie lub focus na polu tekstowym automatycznie otwiera okno kalendarza. Atrybut [*matDatepickerFilter*] przyjmuje funkcję filtrującą, która decyduje, czy dana data jest dostępna do wyboru. W aplikacji landlord-frontend funkcja ta blokuje przedziały dat zajętych przez istniejące najmy, zapobiegając tworzeniu nakładających się umów.

Listing 35: Zastosowanie metody filtrującej w komponencie daty z Angular Material

```
<input
  [(ngModel)]="value"
  [matDatepicker]="datepicker"
  [matDatepickerFilter]="filter"
  [min]="min"
  [max]="max"
  (click)="datepicker.open()"readonly />
<mat-datepicker #datepicker></mat-datepicker>
// Metoda filtrująca
protected filter = (value: string | null): boolean => {
  const date = this.convertDate(value);
  return !this.blockedDateRanges.some(range => {
    const start = this.convertDate(range.start);
    const end = this.convertDate(range.end);
    return date >= start && date <= end;
  });
};
```

Konfiguracja adaptera daty

Angular Material wymaga skonfigurowania tzw. adaptera daty - mechanizmu odpowiadającego za formatowanie wartości dat. Projekt używa adaptera konfiguracyjnego z narzędzia Moment.js (*provideMomentDateAdapter*), który umożliwia pracę z datami w formacie moment.js. Format wyświetlania jest zdefiniowany jako stała

`DATEPICKER_FORMAT_PARSER` w bibliotece `frontend-core` i eksportowana jako część jej publicznego API.

Listing 36: Implementacja konfiguracji daty

```
export const DATEPICKER_FORMAT_PARSER = {
  parse: { dateInput: 'DD.MM.YYYY' },
  display: { dateInput: 'DD.MM.YYYY', monthYearLabel: 'MMM YYYY', ... },
};
```

Stała ta jest następnie przekazywana do konfiguracji aplikacji w `app.config.ts`. Dzięki temu obie aplikacje (`landlord-frontend` i `tenant-frontend`) używają jednolitego formatu daty polskiego `DD.MM.YYYY` bez konieczności duplikowania konfiguracji.

6.10. Implementacja aplikacji backendowych

Implementacja aplikacji backendowych stanowi najbardziej szczegółny etap w procesie tworzenia systemu informatycznego, ponieważ odpowiada za realizację logiki biznesowej, przetwarzanie danych oraz komunikację z warstwą bazodanową i interfejsem użytkownika. W niniejszym projekcie backend został zaprojektowany w sposób modułarny, co umożliwia łatwe utrzymanie, skalowanie oraz rozbudowę systemu w przyszłości.

W trakcie implementacji zastosowano podejście oparte na architekturze mikroserwisowej z podejściem DDD, co pozwoliło na wyraźne rozdzielenie odpowiedzialności pomiędzy poszczególne komponenty systemu. Warstwa kontrolerów oraz ACL odpowiada za obsługę żądań HTTP, walidację danych wejściowych oraz komunikację z warstwą wewnętrzną, która zawiera właściwą logikę biznesową aplikacji. Z kolei warstwa dostępu do danych umożliwia wykonywanie operacji na bazie danych za pomocą dedykowanych repozytoriów.

6.10.1 Współdzielone API dla serwisów

W architekturze mikroserwisowej serwisy muszą wymieniać się między sobą danymi. Dotyczy to zarówno przez wywołania HTTP i kolejki wiadomości. Każde takie połączenie wymaga wspólnego interfejsu. Każdy z serwisów musi wiedzieć jakie pola zawiera obiekt, jakie wartości może przyjmować enum, pod jaką nazwą nasłuchuje kolejka. Bez mechanizmu wymuszającego spójność, istnieje duża podatność na błędy przy każdej zmianie.

Rozwiązaniem zastosowanym w projekcie jest wydzielenie wspólnego interfejsu do osobnych bibliotek Javy. Każdy serwis domenowy ma własny pakiet, który jest dystrybuowany przez prywatne repozytorium Maven w Gitlab. W projekcie funkcjonują 3 główne biblioteki API. Natomiast każdy z serwisów BFF posiada własne dedykowane API, które integruje się z frontendem w dostosowany do niego sposób.

Tabela 3: Przykładowe API zaimplementowane w projekcie

Artefakt	GroupId
rental-service-api	com.rental.platform.project.rental

tenant-service-api	com.rental.platform.project.tenant
authorization-service-api	com.rental.platform.project.authorization

Każda biblioteka jest projektem Maven bez klasy startowej - zawiera wyłącznie klasy modelu DTO, enum, rekordy i stałe. Zależności takie jak Lombok mają zakres (*provided*), co oznacza, że dostarczenie ich spoczywa na serwisie konsumencie.

rental-service-api

Największą z bibliotek jest rental-service-api. Eksponuje kontrakt całej domeny najmu nieruchomości, podzielony na pakiety. Kluczową rolę pełnią klasy zdarzeń. Są to rekordy Javy zawierające, obok danych, nazwę kolejki RabbitMQ jako stałą.

Listing 37: Implementacja współdzielonych struktur dostępnych dla innych serwisów

```
public record PaymentEvent(
    @NonNull Long rentalId,
    @NonNull Long paymentId,
    InvoiceModel invoice
) {
    public static final String SUCCESS_QUEUE_NAME =
        "rental.payment.fees.paid.queue";
}
public record DocumentGenerateEvent() {
    public static final String QUEUE_NAME = "document-generation-queue";
    public static final String DOCUMENT_EXCHANGE_NAME =
        "document-generation-exchange";
}
```

Dzięki temu zarówno producent, jak i konsument kolejki używają tej samej stałej. Dzięki temu zmiana nazwy kolejki wymaga modyfikacji wyłącznie w jednym miejscu i jest natychmiast widoczna jako błąd kompilacji w serwisach konsumenta.

tenant-service-api

Biblioteka eksponuje kontrakt serwisu najemcy. Biblioteka zawiera dla przykładu klika z poniższych struktur:

- TenantDTO - główny obiekt najemcy z polami id, publicId, email, phoneNumber, type, company, individual, user,
- TenantType - enum rozróżniający najemcę indywidualnego od firmowego,
- IndividualDTO i CompanyDTO - dane osobowe i firmowe najemcy,
- UserDTO - powiązane konto użytkownika systemu,
- Oraz wiele innych modeli i struktur danych.

Biblioteka ta jest konsumowana przez serwisy które używają tego konkretnego API, m.in. jest to mikroserwis rental-service, który potrzebuje danych najemcy do konstruowania obiektu najmu.

authorization-service-api

Biblioteka definiuje interfejs dla serwisu autoryzacyjnego. Jej struktura jest celowo ogólna, by mogła być konsumowana przez dowolny serwis chcący mieć API potrzebne do autoryzacji konkretnych danych użytkowników.

Korzyści podejścia

Wydzielenie kontraktów do osobnych bibliotek daje kilka konkretnych gwarancji:

- Spójność typów - obie strony komunikacji operują na identycznych klasach skompilowanych z tego samego źródła, eliminując rozbieżności pól i typów,
- Wersjonowanie - serwis konsument może pozostać na starszej wersji kontraktu podczas gdy producent ewoluuje, co umożliwia niezależne wdrożenia,
- Lokalizacja stałych kolejek - nazwy kolejek i struktur RabbitMQ zdefiniowane są raz, w bibliotece producenta, a nie powielane w konfiguracji każdego serwisu,
- Kompilacyjna weryfikacja zmian - złamanie kontraktu jest od razu widoczne jako błąd kompilacji w serwisach konsumentach, zanim dojdzie do środowiska uruchomieniowego.

6.10.2 BFF - Backend for frontend

Wzorzec *Backend for Frontend (BFF)* polega na stworzeniu dedykowanej warstwy serwerowej dla każdej aplikacji klienckiej. Zamiast udostępniania jednego wspólnego API wszystkim konsumentom. Każdy BFF odpowiada na potrzeby konkretnego klienta. Jego celem jest agregacja danych z wielu serwisów, obsługa uwierzytelnienia, egzekwowanie reguł dostępu i zwracanie danych w formacie dopasowanym do wymagań danego interfejsu. Dzięki temu aplikacja frontendowa nie musi wiedzieć nic o wewnętrznej architekturze systemu, ponieważ komunikuje się wyłącznie z swoim BFF.

Struktura i odpowiedzialności

Oba serwisy są zbudowane w oparciu o Spring Boot 3 i Javę 21. Ich zadaniem jest:

- przyjmowanie żądań HTTP od aplikacji frontendowej,
- weryfikacja tożsamości użytkownika na podstawie tokenu JWT,
- wywołanie odpowiednich serwisów domenowych (np. rental-service, tenant-service) przez RestClient,
- zmapowanie odpowiedzi na własne DTO i zwrócenie ich do klienta,
- Orkiestrowanie operacji pomiędzy kilkoma mikroserwisami.

Każdy serwis posiada własny zestaw kontrolerów REST, pogrupowanych domenowo:

landlord-backend

- LandlordController,
- PropertyController,
- RentalController,
- PaymentController,
- TenantController,
- DocumentController,
- ImageController,
- MeterController,
- UserController,
- VerificationController.

tenant-backend

- RentalController,
- PaymentController,
- DocumentController,
- UserController,
- VerificationController.

Komunikacja z serwisami domenowymi

Każdy BFF komunikuje się z serwisami domenowymi przez HTTP przy użyciu Spring RestClient. Dla każdej domeny istnieje dedykowana klasa, która enkapsuluje szczegóły wywołania URL, metodę, obsługę błędów i logowanie czasu odpowiedzi. Przykładowo *LandlordClient* w landlord-backend wywołuje rental-service w celu rejestracji lub pobrania danych wynajmującego potrzebnych w operacji orkiestrującej pomiędzy mikroservisami.

Listing 38: Przykład obsługi operacji na kliencie serwisu

```
LandlordDTO response = restClient.post()
    .uri(UriComponentsBuilder.fromUriString(rentalServiceUrl)
        .pathSegment("landlords")
        .build()
        .toUriString())
    .body(dto)
    .retrieve()
    .body(LandlordDTO.class);
```

Typy DTO używane w tej komunikacji pochodzą ze współdzielonych bibliotek API (rental-service-api, tenant-service-api), opisanych w rozdziale 6.10.1.

Propagacja tożsamości użytkownika

BFF działa jako serwer zasobów OAuth2, który weryfikuje token JWT wystawiony przez serwis autoryzacyjny. Po weryfikacji identyfikator użytkownika jest przekazywany do

serwisów domenowych jako nagłówek HTTP X-User-Id. Mechanizm ten jest zrealizowany jako interceptor RestClient, który jest skonfigurowany globalnie dla wszystkich wychodzących wywołań. Dzięki temu serwisy domenowe mogą zidentyfikować kontekst użytkownika bez ponownej weryfikacji tokenu.

Listing 39: Implementacja interceptora przekazującego token użytkownika do następnych mikroserwisów

```
private ClientHttpRequestInterceptor accessTokenFromSecurityContextInterceptor() {
    return (request, body, execution) -> {
        String userId =
            SecurityContextHolder.getContext().getAuthentication().getName();
        if (StringUtils.hasText(userId)) {
            request.getHeaders().add("X-User-Id", userId);
        }
        return execution.execute(request, body);
    };
}
```

Kontrola dostępu

Każdy BFF definiuje własną konfigurację uprawnień, dopasowaną do roli danego klienta. W landlord-backend system uprawnień rozróżnia role takie jak **landlord:manager**, **landlord:admin** czy **landlord:readonly**, co pozwala precyzyjnie ograniczyć dostęp do poszczególnych operacji.

Listing 40: Implementacja autoryzacji do określonych endpointów

```
.requestMatchers(HttpMethod.POST, "/api/properties").hasAuthority("SCOPE_landlord:m
anager")
.requestMatchers(HttpMethod.GET, "/api/properties").hasAuthority("SCOPE_landlord:
readonly")
.requestMatchers(HttpMethod.GET, "/api/landlords/users").hasAuthority("SCOPE_land
lord:admin")
```

6.10.3 Authorization Service

Uwierzytelnienia i autoryzacja to kluczowe zagadnienie w systemie wielousługowym. Każdy serwis musi wiedzieć, kto wysłał żądanie i jakie ma uprawnienia. Zamiast implementować te mechanizmy osobno w każdym serwisie, w projekcie wydzielono dedykowany authorization-service. Serwis odpowiada za zarządzanie kontami użytkownika, wystawianie tokenów JWT oraz weryfikację tożsamości podczas procesu rejestracji i logowania.

Technologia i rola serwisu

Authorization-service jest aplikacją Spring Boot 3 zbudowaną na module *spring-boot-starter-oauth2-authorization-server*. Pełni w systemie dwie role jednocześnie:

- serwer autoryzacyjny OAuth - wystawia tokeny JWT podpisane kluczem RSA,
- serwis REST - zarządza kontami użytkowników i procesem weryfikacji.

Serwis przechowuje obecnie dane użytkowników w bazie PostgreSQL przy użyciu Spring Data JPA. Natomiast implementacja jest generyczna, co oznacza możliwą zmianę providera danych użytkowników w dowolnym momencie biznesowym.

Koncepcja Realm oraz kontekst użytkownika

Kluczowym elementem projektu jest Realm. Jest to kontekst, w którym istnieje konto użytkownika. W systemie funkcjonują dwa takie konteksty, odpowiednio są to LANDLORD i TENANT. To samo hasło i adres e-mail mogą istnieć jednocześnie dla obu, ponieważ są to osobne konta. Są one obsługiwane przez osobne aplikacje frontendowe. Podczas logowania klient przekazuje swój kontekst jako część żądania, a *RealmAwareUserDetailsService* wyszukuje użytkownika w bazie uwzględniając tę wartość. Realm jest następnie osadzany jako claim w wystawionym tokenie JWT, co pozwala serwisowi BFF zidentyfikować kontekst użytkownika bez dodatkowych zapytań.

Listing 41: Implementacja szukania użytkowników razem z kontekstem

```
UserEntity userEntity = userRepository
    .findByUsernameAndRealm(username, realm)
    .orElseThrow(() -> new UsernameNotFoundException(...));
```

Po zalogowaniu serwis rozwiązuje *contextId*, który jest identyfikatorem encji domenowej powiązanej z użytkownikiem. Dla kontekstu LANDLORD jest to *publicId* konta wynajmującego. Dla TENANT jest to *publicId* konta najemcy. Logika ta jest zawarta w *ContextServiceImpl*.

Listing 42: Implementacja pobierania kontekstu

```
case LANDLORD ->
landlordUserService.getLandlordIdByUserPublicId(userPrincipal.getPublicId())
case TENANT -> entityManager.createNativeQuery(query(realm))
    .setParameter("userId", userPrincipal.getId())
    .getSingleResult().toString()
```

ContextId trafia do tokenu JWT i jest odczytywany przez serwisy BFF, które używają go do filtrowania danych. Przykładem jest np. *landlord-backend*, który pobiera tylko nieruchomości i umowy powiązane z danym kontem wynajmującego.

Wystawianie tokenów JWT

Tokeny są podpisywane algorytmem *RSA-256*. Klucz prywatny jest przechowywany w Java KeyStore (.jks) i ładowany przy starcie aplikacji. Na jego podstawie tworzony jest JWKSsource, który jest następnie używany zarówno do podpisywania tokenów jak i ich odpowiedniej weryfikacji.

Listing 43: Implementacja klucza do podpisywania tokenów

```
RSAPublicKey jwk = new RSAPublicKey.Builder((RSAPublicKey) entries.getPublic())
    .privateKey((RSAPrivateKey) entries.getPrivate())
    .keyID(kid)
    .algorithm(JWSAlgorithm.RS256)
    .build();
```

Token dostępowy zawiera następujące claimy

- sub (*publicId* użytkownika),
- username, realm,
- contextId (*publicId* zarządcy lub najemcy),
- scope (uprawnienia),
- iss,
- iat,
- exp.

Czas ważności tokenu wynosi 3600 sekund (1 godzina). Serwis wystawia również tymczasowy token z zakresem **password:set**. Token ten jest ważny 300 sekund, używany wyłącznie do jednorazowego ustawienia hasła przez nowo dodanego użytkownika.

Proces rejestracji i weryfikacji

Rejestracja użytkownika z poziomu wynajmującego przebiega w kilku krokach. Nowe konto tworzone jest w stanie konta nieaktywnego. Następnie serwis generuje kod weryfikacyjny, który jest wysyłany do użytkownika. Dopiero po poprawnej weryfikacji konto zostaje aktywowane i użytkownik może ustawić hasło przy użyciu tymczasowego tokenu. Serwis weryfikacyjny obsługuje dwa rodzaje kodów:

- OTP_CODE - 6-cyfrowy kod numeryczny, ważny 15 minut, przechowywany jako hash SHA-256,
- EMAIL_CODE - 32-znakowy kod losowy, ważny 24 godziny, przechowywany w postaci jawnej;

Kod OTP jest weryfikowany przez porównanie hashy z użyciem *MessageDigest.isEqual*, co chroni przed atakami czasowymi. Każdy kod posiada licznik pozostałych prób, który jest dekrementowany przy każdej niepoprawnej weryfikacji.

Uprawnienia

Uprawnienia użytkownika są przechowywane w tabeli authority jako wartości enum powiązane relacją *@OneToMany* z encją użytkownika. Nowo zarejestrowany zarządca otrzymuje domyślnie pełen zestaw uprawnień.

Listing 44: Lista ról przypisana do głównego zarządcy

```
private static final Set<Authority> SUPER_USER_DEFAULT_LANDLORD_AUTHORITIES =
Set.of(
    Authority.LANDLORD_ADMIN_ROLE,
    Authority.LANDLORD_MANAGER_ROLE,
    Authority.LANDLORD_READ_ONLY_ROLE,
    Authority.LANDLORD_USER_ROLE,
    Authority.LANDLORD_TENANT_RELATIONS_ROLE
);
```

6.10.4 Generowanie dokumentów

W mikroservisie document-service narzędzie *PuppeteerSharp* odpowiada za generowanie dokumentów PDF, takich jak umowy najmu i faktury. Dokumenty tworzone są na podstawie szablonów HTML wypełnianych danymi przekazanymi w momencie żądania.

Inicjalizacja przeglądarki

Przeglądarka jest inicjalizowana raz przy starcie serwisu. *BrowserFetcher* automatycznie pobiera odpowiednią wersję Chromium, a *Puppeteer.LaunchAsync()* uruchamia ją w trybie Headless.

Listing 45: Implementacja inicjalizacji narzędzia do generowania dokumentów
PuppeteerSharp

```
private async Task IntitalizeBrowser()
{
    var browserFetcher = new BrowserFetcher();
    await browserFetcher.DownloadAsync();

    _browser = await Puppeteer.LaunchAsync(new LaunchOptions
    {
        Headless = true,
        Args = new[] { "--no-sandbox", "--disable-setuid-sandbox" }
    });
}
```

Generowanie PDF

Metoda *GeneratePdf* dostaje szablon HTML oraz odpowiednie dane przygotowane pod konkretny z nich, następnie ładuje do niej gotowy HTML i eksportuje do PDF z konfiguracją formatu A4. Implementacja pokazana została na listingu 46.

Listing 46: Implementacja metody generującej dokument PDF

```
public async Task<byte[]> GeneratePdf(string szablon, object dane)
{
    var template = Template.Parse(szablon);
    var html = await template.RenderAsync(dane);

    await using var page = await _browser.NewPageAsync();
    await page.SetContentAsync(html);

    var pdfBytes = await page.PdfDataAsync(new PdfOptions
    {
        Format = PaperFormat.A4,
        PrintBackground = true,
        MarginOptions = new MarginOptions
        {
            Top = "20mm", Bottom = "20mm",
            Left = "15mm", Right = "15mm"
        }
    });

    return pdfBytes;
}
```

6.10.5 Mechanizm streamingu i double batch

W systemach przetwarzających duże ilości danych często jest stosowane nieoptymalne podejście. Tym rozwiązaniem jest wczytanie wszystkich rekordów do pamięci i przetworzenie ich w jednej pętli prowadzi do nadmiernego zużycia zasobów. W projekcie, w kontekście cyklicznego przetwarzania umów najmu, zastosowano dwa mechanizmy rozwiązujące ten problem: streaming wyników z bazy danych oraz batch processing.

Streaming wyników z bazy danych

Zamiast pobierać wszystkie rekordy naraz jako listę, repozytorium przekazuje stream poprzez interface *Stream<RentalEntity>*. Adnotacja *@QueryHint* nałożona na metodę w repozytorium z wartością *HINT_FETCH_SIZE = 1000* oraz odpowiednia konfiguracja instruuje sterownik JDBC, aby pobierał rekordy z bazy porcjami po 1000 wierszy zamiast wczytywać cały wynik zapytania do pamięci jednorazowo. Hibernate utrzymuje otwarte połączenie z bazą i pobiera kolejne porcje w miarę konsumowania strumienia.

Listing 47: Implementacja metody potrzebnej do streamowania danych z bazy

```
@QueryHints(@QueryHint(name = HINT_FETCH_SIZE, value = "1000"))
Stream<RentalEntity> findByStatusAndEndDate(
    @NonNull RentalStatus status,
```

```

        @NonNull LocalDate endDate
    );

```

Strumień jest otwierany w bloku try-with-resources, co gwarantuje zamknięcie kursora bazy danych nawet w przypadku wyjątku. Każdy rekord jest odłączany od sesji JPA przez *entityManager.detach()*, co pozwala na bezproblemowe kontrolowanie tej operacji. Bez tego wywołania każdy wczytany obiekt pozostawałby w pamięci podręcznej pierwszego poziomu Hibernate przez cały czas trwania transakcji. Sprawiałoby to, że przy tysiącach rekordów prowadziłoby do wyczerpania pamięci. Rekordy są następnie filtrowane i grupowane według identyfikatora najemcy.

Listing 48: Użycie metody streamującej z bazy danych, w klauzuli try

```

try (Stream<RentalEntity> entitiesStream =
    repository.findByStatusAndEndDate(RENTAL_ACTIVE, endDate)) {
    entitiesStream
        .filter(entity -> entity.getEndDate() != null && !entity.isIndefinite())
        .peek(entityManager::detach)
        .forEach(entity -> map.computeIfAbsent(
            entity.getTenantId(),
            k -> new ArrayList<>()).add(entity)
        );
}

```

Cache wynajmujących potrzebny do optymalizacji batchy

Dane wynajmujących są buforowane w pamięci cache Caffeine, która jest skonfigurowana na maksymalnie 10 000 wpisów z wygaśnięciem po 30 minutach. Przed każdym wywołaniem zewnętrznym sprawdzany jest cache. Identyfikatory nieobecne w pamięci podręcznej trafiają faktycznie do zapytania.

Listing 49: Użycie cache do optymalizacji operacji

```

@Bean
Cache<Long, Landlord> landlordCache() {
    return Caffeine.newBuilder()
        .maximumSize(10_000)
        .expireAfterWrite(Duration.ofMinutes(30))
        .build();
}

// użycie cache
for (Long id : requiredLandlordIds) {
    Landlord cached = landlordCache.getIfPresent(id);
    if (cached != null) {
        landlordMap.put(id, cached);
    }
}

```

```

    } else {
        idsToFetch.add(id);
    }
}

if (!idsToFetch.isEmpty()) {
    List<Landlord> fetched = landlordService.getBaseByIds(idsToFetch);
    fetched.forEach(l -> {
        landlordCache.put(l.getId(), l);
        landlordMap.put(l.getId(), l);
    });
}

```

6.10.6 Mechanizm mierników zasobów

System rejestruje zużycie mediów dla każdej nieruchomości w postaci pięciu mierników:

- energii elektrycznej (*ELECTRICITY*),
- zimnej wody (*WATER_COLD*),
- ciepłej wody (*WATER_HOT*),
- gazu (*GAS*),
- ogrzewania (*HEATING*).

Energia elektryczna i ogrzewanie są mierzone w kilowatogodzinach, natomiast pozostałe media w metrach sześciennych. Każdy miernik jest przechowywany w bazie danych jako osobny rekord powiązany z nieruchomością. Operacja *upsert* przy pierwszym wywołaniu tworzy komplet mierników dla wszystkich typów, przy kolejnych aktualizuje ich wartości.

Listing 50: Implementacja wyboru odpowiedniego typu dla licznika

```

private UnitType getUnitTypeForMeter(@NonNull MeterType type) {
    return switch (type) {
        case ELECTRICITY, HEATING -> UnitType.KWH;
        case WATER_COLD, WATER_HOT, GAS -> UnitType.CUBIC_METER;
    };
}

```

Operacja *upsert* sprawdza najpierw czy dla danej nieruchomości istnieją już rekordy. Jeśli nie, tworzy je dla wszystkich typów jednocześnie. Wszystkie rekordy są zapisywane jednym wywołaniem *saveAll*.

Listing 51: Implementacja zapisu nowych mockowanych wartości licznika

```

@Transactional
public void upsert(@NonNull Long propertyId) {
    List<MeterEntity> entities = repository.findByPropertyId(propertyId);
}

```

```

if (entities.isEmpty()) {
    entities = Arrays.stream(MeterType.values())
        .map(type -> buildEntity(type, propertyId))
        .toList();
} else {
    entities.forEach(entity ->
        entity.setValue(generateValue(entity.getValue())));
}

repository.saveAll(entities);
}

```

6.10.7 Obsługa asynchroniczna wielu wątków

W systemie istnieje kilka miejsc, w których możliwy jest potencjalny duży nakład operacji w jednym momencie, takich jak wysyłka powiadomień, czy obsługa automatycznych zmian statusów. Odpowiedzialnymi mechanizmami za to są moduły planowania, czyli tak zwane schedulery. Obsługują zdarzenia cyklu życia najmu (przypomnienie, zakończenie umów, start najmów) i są przystosowane do przetwarzania wielu rekordów w partiach jednym czasie. Zapobiega to nadmiernemu zużyciu pamięci przy jednoczesnym przetwarzaniu dużej liczby rekordów oraz chroni zewnętrzne serwisy przed nagłym przyrostem zapytań.

Listing 52: Przetwarzanie danych partiami, wykorzystując interfejs `CompletableFuture` do operacji asynchronicznych

```

private static final Integer BATCH_SIZE = 100;
private void sendRentalNotifications(
    Map<Long, List<RentalEntity>> tenantRentalsMap,
    EmailType emailType
) {
    List<Long> tenantIds = new ArrayList<>(tenantRentalsMap.keySet());

    for (int i = 0; i < tenantIds.size(); i += BATCH_SIZE) {
        List<Long> batchIds = tenantIds.subList(
            i, Math.min(i + BATCH_SIZE, tenantIds.size()));
        CompletableFuture.runAsync(
            () -> processNotification(
                batchIds,
                tenantRentalsMap,
                emailType
            ), taskExecutor
        );
    }
}

```

6.10.8 Optymalizacja requestów poprzez batchowe odpytanie danych

Buforowanie jest przede wszystkim w komponentach przetwarzających duże wolumeny danych np. Schedulerach i listenerach zdarzeń płatniczych. Dane wynajmujących i najemców, które są wielokrotnie odpytywane w trakcie przetwarzania, pobierane są z zewnętrznych serwisów tylko raz, a następnie serwowane z lokalnego cache'u dla kolejnych żądań w ramach tego samego cyklu. Eliminuje to wielokrotne wywołania sieciowe dla tych samych danych.

Listing 53: Implementacja buforowania danych z innego mikroserwisu przy pomocy cache'owania

```
private Map<Long, Tenant> fetchTenants(Map<Long, Rental> rentalsById) {
    Map<Long, Tenant> result = new HashMap<>();
    List<Long> toFetch = new ArrayList<>();

    for (Rental rental : rentalsById.values()) {
        Long tenantId = rental.getTenantId();
        Tenant cached = tenantCache.getIfPresent(tenantId);
        if (cached != null) {
            result.put(tenantId, cached);
        } else {
            toFetch.add(tenantId);
        }
    }

    if (!toFetch.isEmpty()) {
        tenantService.getByPrivateIds(toFetch).forEach(t -> {
            tenantCache.put(t.getPrivateId(), t);
            result.put(t.getPrivateId(), t);
        });
    }

    return result;
}
```

6.10.9 Mechanizm obsługi powiadomień

Producent zdarzenia (np. rental-service po zawarciu umowy) publikuje komunikat do kolejki i natychmiast kontynuuje działanie, nie blokując się na czas przetwarzania. Konsument (np. notification-service) odbiera wiadomości w swoim tempie, niezależnie od producenta. Drugim przykładem jest implementacja reagowania na zdarzenia dziejące się na encjach w bazie takich jak *AFTER_COMMIT*, gdzie po takiej akcji publikowane jest wewnętrzne zdarzenie, które możemy przejąć konkretną metodą serwisową i obsłużyć dalsze operacje. Taki model pozwala na swobodne skalowanie konsumentów przez dodawanie kolejnych instancji nasłuchujących na tej samej kolejce.

Listing 54: Implementacja operacji obsługi zdarzenia oraz odbioru

```

record RentalCreatedEvent(
    @NonNull Rental rental,
    @NonNull Tenant tenant,
    @NonNull Property property,
    @NonNull Landlord landlord
) { }

// Publikowanie przez RentalServiceImpl po zapisaniu umowy
RentalEntity savedEntity = repository.save(entity);
feeService.initial(rental.getFees(), savedEntity.getId(),
    savedEntity.getStartDate());
eventPublisher.publishEvent(new RentalCreatedEvent(
    factory.build(savedEntity, rental.getFees()),
    tenant, property, landlord
));
// Wysyłka odpowiednich danych i informacji na kolejkę do document-service oraz
notification-service po poprawnym zapisie na bazie
@TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
public void handleRentalInitializationSuccess(@NonNull RentalCreatedEvent event) {
    emailService.send(EmailImpl.builder()
        .rental(event.rental())
        .tenant(event.tenant())
        .property(event.property())
        .type(EmailType.RENTAL_AGREEMENT_CREATED)
        .build());

    documentService.send(DocumentImpl.builder()
        .landlord(event.landlord())
        .property(event.property())
        .tenant(event.tenant())
        .rental(event.rental())
        .type(DocumentType.RENTAL_AGREEMENT)
        .build());
}

```

6.11. DTO

W komunikacji między warstwami aplikacji np. między frontendem, a BFF lub między BFF, a serwisami domenowymi dane są przesyłane w postaci obiektów DTO. Są to klasy zawierające wyłącznie pola danych, bez logiki biznesowej. Ich zadaniem jest przeniesienie danych przez granicę warstwy w określonym kształcie, niezależnie od wewnętrznej reprezentacji encji domenowych.

W projekcie każdy serwis definiuje własne DTO dopasowane do swoich potrzeb. Przykładowo RentalDTO w landlord-backend zawiera istotne dla widoku wynajmującego. W

tenant-backend RentalDTO zawiera pola, które są istotne dla najemcy, mimo że oba opisują ten sam obiekt domenowy. Dzięki temu rozwiązaniu każda warstwa może się rozwijać niezależnie, bez wpływu na inne serwisy.

Klasy DTO w części backendowej są budowane przy użyciu adnotacji Lombok takich jak `@Data`, `@Builder`, `@NoArgsConstructor`, `@AllArgsConstructor`, co eliminuje niepotrzebny kod związany z getterami, seterami i konstruktorami. Adnotacja `@JsonInclude(JsonInclude.Include.NON_NULL)` powoduje, że pola o wartości null są pomijane podczas serializacji, co zmniejsza rozmiar odpowiedzi i upraszcza obsługę opcjonalnych pól po stronie klienta.

6.12. JSON

Format JSON, czyli (*JavaScript Object Notation*) jest jedynym formatem wymiany danych stosowanym w komunikacji HTTP w projekcie. Serializację i deserializację obiektów Java do i z JSON obsługuje biblioteka Jackson, integrowana automatycznie przez Spring Boot. Po stronie Angular komunikacja z API odbywa się przez HttpClient, który domyślnie serializuje obiekty TypeScript do JSON i deserializuje odpowiedzi do typowanych interfejsów.

6.13. Część wdrożeniowa systemu

Po zakończeniu prac związanych z projektowaniem, implementacją oraz testowaniem system został przygotowany do wdrożenia. Etap ten miał na celu zweryfikowanie możliwości wykorzystania opracowanego rozwiązania w warunkach zbliżonych do rzeczywistego środowiska pracy.

W pierwszej kolejności przeprowadzono proces uruchomienia systemu z wykorzystaniem docelowej konfiguracji. Pozwoliło to na sprawdzenie współpracy wszystkich modułów oraz ocenę stabilności działania aplikacji podczas wykonywania typowych operacji przez użytkowników. Szczególną uwagę zwrócono na poprawność przepływu danych pomiędzy poszczególnymi komponentami systemu.

Wdrożenie umożliwiło również ocenę korzyści wynikających z zastosowania systemu. W porównaniu z dotychczasowym sposobem realizacji procesów rozwiązanie pozwala na skrócenie czasu wykonywania wybranych operacji, ograniczenie liczby błędów wynikających z ręcznego przetwarzania danych oraz poprawę dostępności informacji. Dodatkowo użytkownicy otrzymali narzędzie umożliwiające szybszy dostęp do danych i bardziej efektywne zarządzanie realizowanymi zadaniami.

6.13.1 Lokalne środowisko

Lokalne środowisko deweloperskie opiera się na narzędziu Docker oraz Docker Compose. Każdy serwis backendowy posiada własny plik `docker-compose.yml`, który uruchamia go jako kontener z obrazu pobranego z Gitlab Container Registry. Wszystkie kontenery są podłączone do wspólnej sieci Docker o nazwie `rental_platform_network`. Umożliwia to komunikację między serwisami przez nazwy kontenerów zamiast adresów IP.

Listing 55: Konfiguracja docker-compose jednego z serwisów

```

services:
  landlord-backend-service:
    image: registry.gitlab.com/rental-platform-project/landlord-backend:latest
    environment:
      SERVICES_RENTAL_SERVICE_URL:      http://rental-service:8082
      SERVICES_TENANT_SERVICE_URL:      http://tenant-service:8087
      SERVICES_PAYMENT_SERVICE_URL:     http://payment-service:8083
      SERVICES_DOCUMENT_SERVICE_URL:    http://document-service:5219
      SERVICES_AUTHORIZATION_SERVICE_URL: http://authorization-service:8081
    networks:
      - rental_platform_network
    ports:
      - "9080:9080"

```

Baza danych PostgreSQL 17 oraz RabbitMQ uruchamiane są osobnym plikiem *compose.yml* z projektu devops-rental, który tworzy współdzielony wolumin danych i sieć używaną przez pozostałe serwisy.

Aplikacje frontendowe (landlord-frontend, tenant-frontend) w środowisku lokalnym są uruchamiane bezpośrednio przez Angular CLI (ng serve) na portach 4200, z konfiguracją proxy kierującą żądania API do odpowiednich serwisów BFF lub mogą zostać uruchomione jako dockerowe kontenery głównego silnika nginx.

6.13.2 Środowisko produkcyjne

Środowisko produkcyjne jest oparte na klastrze k3s, który jest lekką dystrybucją Kubernetes przeznaczoną dla środowisk o ograniczonych zasobach. Każdy serwis jest wdrożony jako oddzielny zasób Kubernetes typu Deployment, Service i Ingress. Obrazy Docker są budowane przez potok CI/CD w GitLab i publikowane do GitLab Container Registry. Każdy serwis posiada *.gitlab-ci.yml* z dwoma etapami: build (budowanie obrazu Docker i push do registry) oraz deploy (aplikowanie manifestów Kubernetes przez kubectl). Wdrożenie uruchamiane jest automatycznie przy każdym pushu na branch master. Przykładowy fragment potoku CI/CD dla landlord-frontend pokazany na liście 56.

Listing 56: Konfiguracja automatyzacji Gitlab

```

build:
  script:
    - docker build --build-arg GITLAB_ACCESS_TOKEN=${GITLAB_ACCESS_TOKEN}
      -t $CI_REGISTRY_IMAGE:latest -f docker/Dockerfile .
    - docker push $CI_REGISTRY_IMAGE:latest
deploy:
  script:
    - kubectl apply -f k3s/Deployment.yml
    - kubectl apply -f k3s/Service.yml
    - kubectl apply -f k3s/Ingress.yml
    - kubectl rollout restart deployment landlord-frontend

```

6.13.3 Przekierowanie ruchu na poszczególne domeny

Ruch HTTP w klastrze k3s jest zarządzany przez Traefik, który jest kontrolerem Ingress wbudowanym domyślnie w k3s. Każdy serwis posiada osobny zasób Ingress, który definiuje reguły routingu na podstawie prefiksu ścieżki URL.

Tabela 4: Tabela przedstawiająca ingressy w poszczególnych serwisach

Ścieżka	Serwis docelowy	Port
/landlord	landlord-frontend-service	80
/tenant	tenant-frontend-service	80
/landlord/api	landlord-backend	9080
/tenant/api	tenant-backend	7080

Serwisy backendowe korzystają z techniki middleware Traefik *strip-prefix*, który usuwa w przypadku systemu prefiks ścieżki (/landlord lub /tenant) przed przekazaniem żądania do kontenera. Dzięki temu serwis backendowy widzi ścieżkę /api/... zamiast /landlord/api/..., co eliminuje potrzebę konfigurowania prefiksu wewnątrz aplikacji.

Aplikacje frontendowe są budowane z parametrem `--base-href /landlord/` (lub `/tenant/`), co informuje Angular o bazowej ścieżce aplikacji i pozwala prawidłowo generować linki oraz ładować zasoby statyczne.

6.13.4 Podsumowanie

Środowisko lokalne oparte na *Docker Compose* pozwala na uruchomienie całego systemu na pojedynczej maszynie deweloperskiej bez konieczności dostępu do infrastruktury produkcyjnej. Każdy serwis jest izolowany w osobnym kontenerze, a komunikacja między nimi odbywa się przez wspólną sieć Docker z użyciem nazw kontenerów jako hostów.

Środowisko produkcyjne oparte na k3s zapewnia te same gwarancje izolacji i konfigurowalności co pełny Kubernetes. Automatyczne wdrożenia przez *GitLab CI/CD* eliminują ręczne kroki. Każdy push do brancha master kończy się zbudowaniem nowego obrazu oraz uruchomieniem nowej wersji serwisu w klastrze. Dane wrażliwe są zarządzane przez *Kubernetes Secrets*, co oddziela konfigurację środowiskową od kodu aplikacji.

Routing ruchu przez Traefik Ingress pozwala na udostępnienie wszystkich serwisów pod jednym adresem IP klastra, z podziałem na ścieżki URL. Takie podejście upraszcza konfigurację sieciową i eliminuje potrzebę zarządzania osobnymi adresami dla każdego serwisu.

7. Interfejs użytkownika

Interfejs użytkownika to najważniejsza część systemu informatycznego osób końcowych, która odpowiada za sposób komunikacji między człowiekiem, a urządzeniem lub oprogramowaniem. To właśnie poprzez interfejs użytkownik może wydawać polecenia, odbierać informacje zwrotne oraz wchodzić w interakcję z aplikacją w sposób zrozumiały i intuicyjny.

Współczesne interfejsy użytkownika odgrywają kluczową rolę w projektowaniu oprogramowania, ponieważ decydują o tym, jak łatwo i wygodnie można korzystać z danego systemu. Nawet najbardziej zaawansowana aplikacja może okazać się nieprzydatna, jeśli jej obsługa jest nieintuicyjna lub zbyt skomplikowana. Dlatego projektowanie UI łączy w sobie elementy informatyki, psychologii oraz designu.

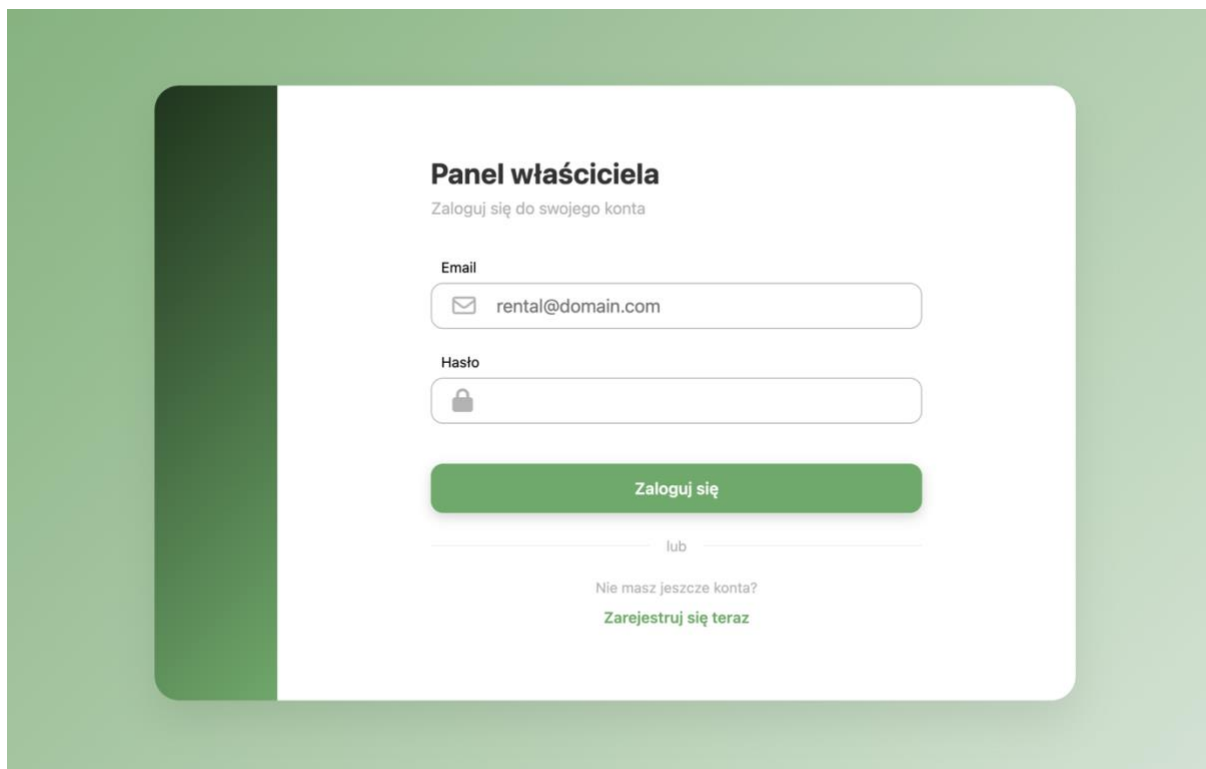
Celem interfejsu użytkownika jest zapewnienie jak największej użyteczności, przejrzystości oraz komfortu pracy. Obejmuje on zarówno elementy wizualne, takie jak przyciski, menu czy ikony, jak i sposób reagowania systemu na działania użytkownika. Najważniejszymi rzeczami na tym etapie jest spójne oraz optymalne pokazywanie odpowiednich ekranów użytkownikowi, żeby mógł poczuć się jako osoba, która ma kontrolę nad aplikacją bez zbędnych i niepotrzebnych przeszkód, które osoby.

7.1. Aplikacja webowa zarządcy

Opisane w poprzednich rozdziałach podziały domenowe systemu znajdują swoje odzwierciedlenie w widokach prezentowanych na ekranach dla użytkownika, które zostały przedstawione w tym rozdziale. Jedną z domen systemu jest domena zarządcy nieruchomości. W dalszej części podrozdziału przedstawiono widoki systemowe dostępne dla użytkownika posiadającego rolę zarządcy wraz z opisem ich funkcjonalności.

7.1.1 Ekran logowania

Ekran logowania pokazany na rysunku 16 pojawia się użytkownikowi podczas wejścia bezpośrednio na pusty URL tzw. (ukośnik), skąd Angular przekierowuje takiego użytkownika odpowiednio na URL `/auth/login`. Na tym ekranie użytkownik jest w stanie zalogować się do swojego konta bądź, jeśli takiego jeszcze nie posiada, spróbować zarejestrować się w panelu klikając na przycisk „Zarejestruj się teraz”.



Rysunek 16: Ekran logowania do panelu zarządcy

7.1.2 Ekran rejestracji i logowania zarządcy głównego

Proces rejestracji nowego konta właściciela został podzielony na trzy kroki, wizualnie za pomocą paska postępu widocznego w górnej części formularza. Taki podział ogranicza liczbę pól prezentowanych jednocześnie użytkownikowi. Rozwiązanie to upraszcza wypełnienie formularza i zmniejsza ryzyko błędów oraz ознајmia użytkownika, ile kroków zostało do końca. Ogranicza to potencjalny brak pewności użytkownika w kontekście czasu jaki musi poświęcić do końca uzupełnienia formularza.

W pierwszym kroku użytkownik podaje dane kontaktowe organizacji:

- adres e-mail,
- numer telefonu,
- wyświetlaną nazwę.

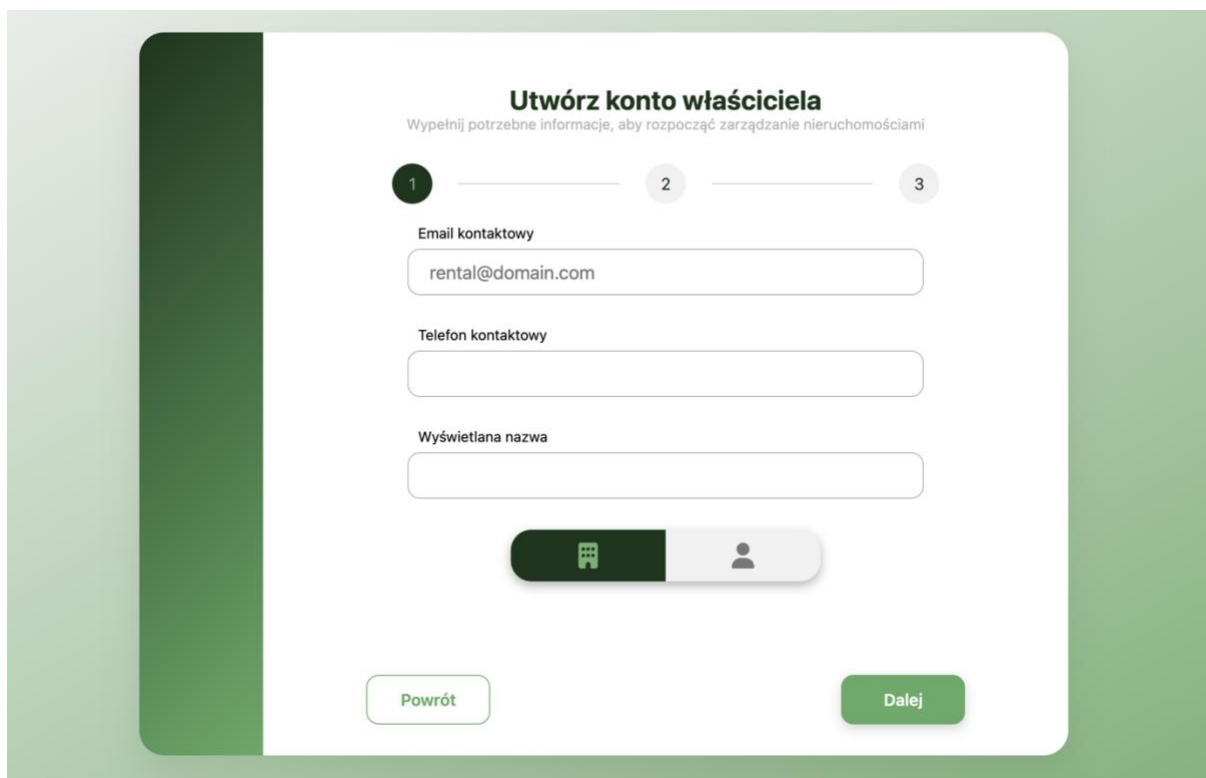
Dostępny jest również przełącznik umożliwiający wybór typu konta - firmowego lub prywatnego, wpływa on na zakres danych wymaganych w kolejnym kroku.

Drugi krok obejmuje dane adresowe i podatkowe. Identyfikator podatkowy (NIP), kraj, miasto, ulicę, numer mieszkania oraz kod pocztowy. Na podstawie wprowadzonego identyfikatora podatkowego system może zweryfikować podmiot w rejestrze GUS.

W trzecim i ostatnim kroku użytkownik ustala dane logowania. Adres e-mail przypisany do konta oraz hasła. Po kliknięciu przycisku „Założ konto” konto zostaje utworzone, a użytkownik uzyskuje dostęp do panelu zarządcy.

Nawigacja między krokami odbywa za pomocą przycisków „Dalej” i „Powrót”, co umożliwia swobodne cofnięcie się i poprawienie wcześniej wprowadzonych danych bez utraty wypełnionych pól. Na rysunkach od 17 do 20 zostały pokazane wszystkie kroki rejestracji takiego zarządcy.

Ekran kroku pierwszego



The screenshot shows a mobile application interface for creating a landlord account. The title is "Utwórz konto właściciela" (Create landlord account). Below the title is a subtitle: "Wypełnij potrzebne informacje, aby rozpocząć zarządzanie nieruchomościami" (Fill in the necessary information to start managing properties). There is a progress indicator at the top with three steps: 1 (active), 2, and 3. The first step contains three input fields: "Email kontaktowy" (Contact email) with the value "rental@domain.com", "Telefon kontaktowy" (Contact phone), and "Wyświetlana nazwa" (Display name). Below the input fields is a toggle switch with a house icon on the left and a person icon on the right. At the bottom are two buttons: "Powrót" (Back) and "Dalej" (Next).

Rysunek 17: Ekran pierwszego kroku podczas rejestracji zarządcy

Ekran kroku drugiego

Utwórz konto właściciela
Wypełnij potrzebne informacje, aby rozpocząć zarządzanie nieruchomościami

1 — 2 — 3

Identyfikator podatkowy

Kraj

Miasto

Ulica

Numer mieszkania

Kod pocztowy

Powrót Dalej

Rysunek 18: Ekran drugiego kroku podczas rejestracji zarządcy dla konta firmowego

Utwórz konto właściciela
Wypełnij potrzebne informacje, aby rozpocząć zarządzanie nieruchomościami

1 ————— 2 ————— 3

Imię

Nazwisko

Numer pesel

Kraj

Miasto

Ulica

Numer mieszkania

Kod pocztowy

[Powrót](#) [Dalej](#)

Rysunek 19: Ekran kroku drugiego podczas rejestracji zarządcy dla konta indywidualnego

Ekran kroku trzeciego

Utwórz konto właściciela

Wypełnij potrzebne informacje, aby rozpocząć zarządzanie nieruchomościami

1 2 3

Email do konta

test@test.com

Hasło

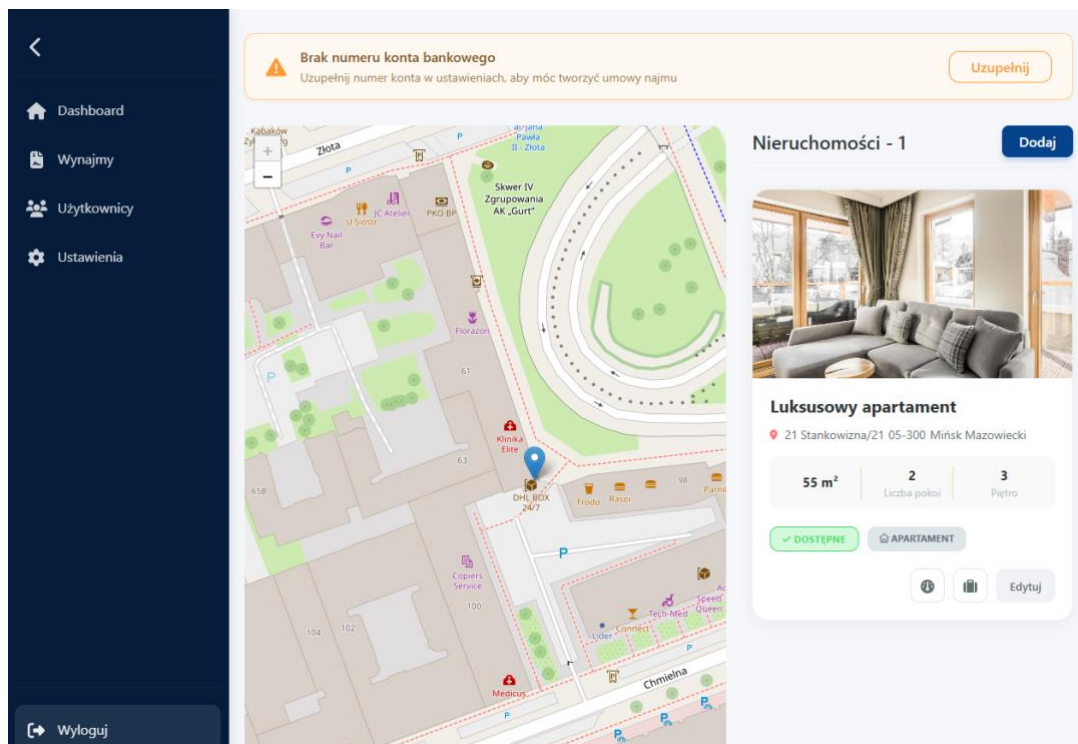
Powrót Załącz konto

Rysunek 20: Ekran trzeci podczas rejestracji zarządcy

7.1.3 Ekran główny

Ekran główny aplikacji zarządcy pokazany na rysunku 21 składa się z dwóch głównych obszarów: bocznego menu nawigacyjnego oraz przestrzeni roboczej prezentującej nieruchomości przypisane do konta.

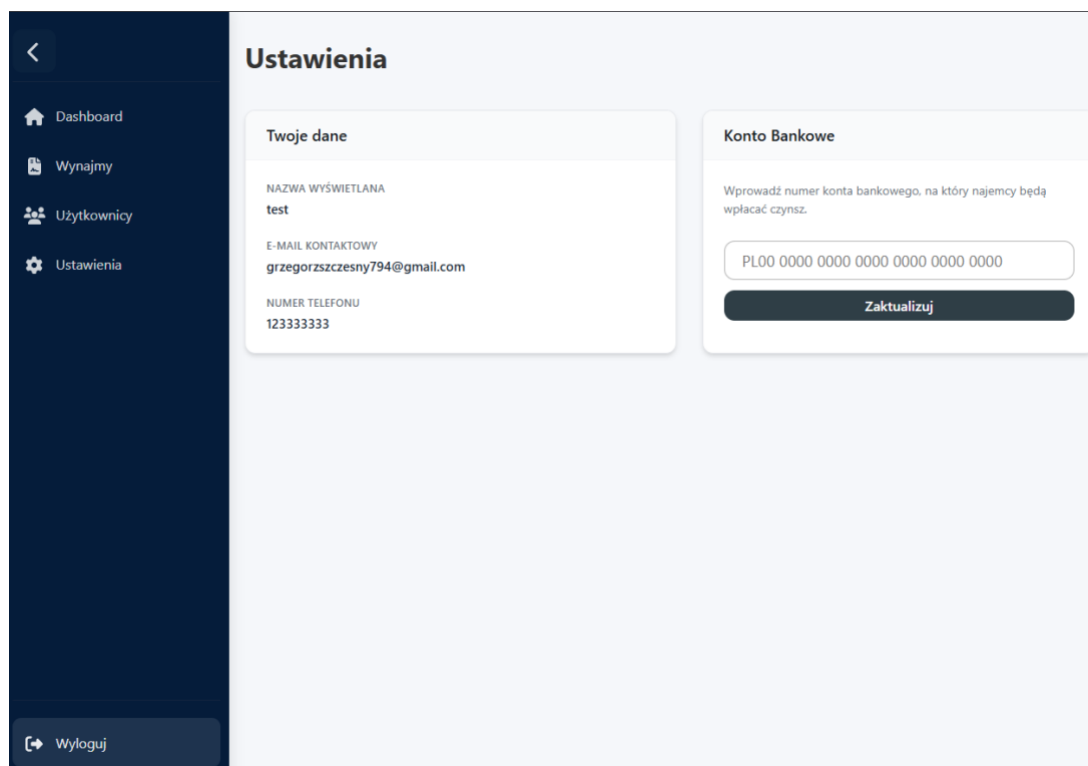
Lewy panel nawigacyjny zawiera odnośniki do modułów systemu: Dashboard, Wynajmy, Użytkownicy oraz Ustawienia. W dolnej części przycisk wylogowania. Menu może być zwijane w celu powiększenia przestrzeni roboczej.



Rysunek 21: Panel główny w aplikacji zarządcy

Główna część ekranu podzielona jest na dwa widoki. Po lewej stronie wyświetlana jest interaktywna mapa z zaznaczonymi lokalizacjami nieruchomości zarządcy. Każdy punkt na mapie odpowiada konkretnemu lokalowi. Po prawej stronie prezentowana jest lista kart nieruchomości, gdzie każda karta zawiera zdjęcie lokalu, jego nazwę, adres, powierzchnię w metrach kwadratowych, liczbę sypialni, piętro, a także status dostępności i typ nieruchomości. Dostępne są również przyciski umożliwiające wykonanie różnych operacji związanych na danym lokalu, m.in. podgląd szczegółów czy edycję lokalu. Przycisk „Dodaj” w prawym górnym umożliwia dodanie nowej nieruchomości.

7.1.4 Ekran ustawień



Rysunek 22: Ekran ustawień w panelu zarządcy

Ekran ustawień przedstawiony na rysunku 22 umożliwia podgląd i konfigurację podstawowych danych swojego konta. Lewa karta „Twoje dane” wyświetla w trybie tylko do odczytu informacje wprowadzone podczas rejestracji: nazwę wyświetlaną organizacji, kontaktowy adres e-mail oraz numer telefonu. Prawa karta „Konto Bankowe” umożliwia wprowadzenie lub aktualizację numeru rachunku bankowego, na który najemcy będą dokonywać płatności czynszowych.

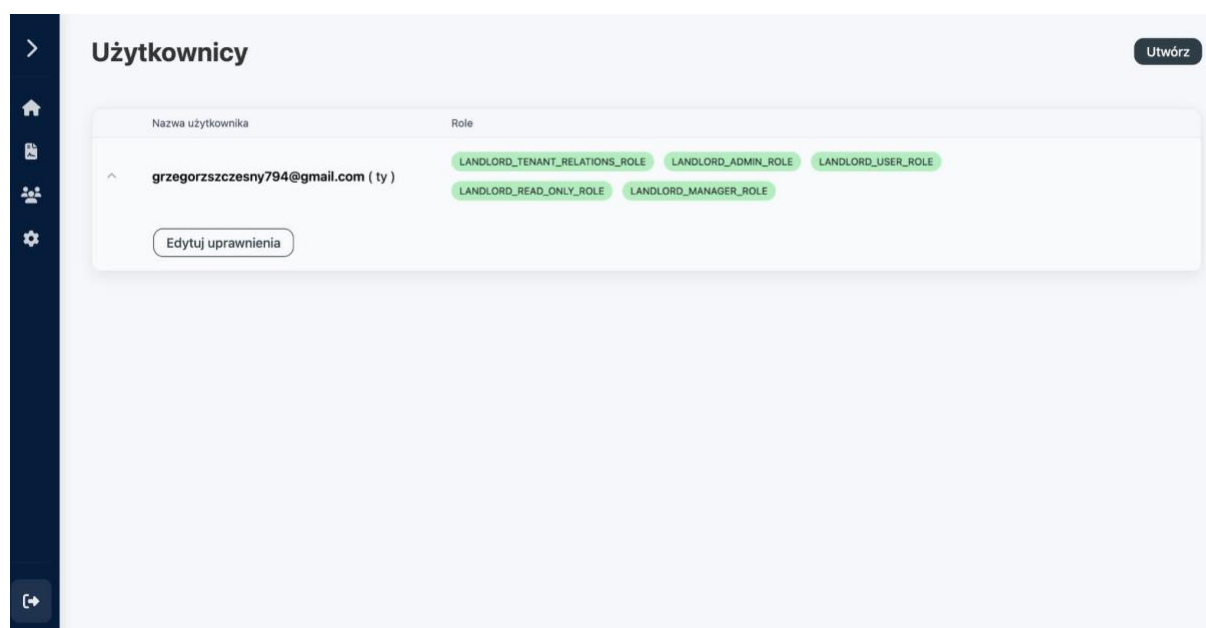
7.1.5 Ekran kreatora najmu

Ekran kreatora najmu, składający się z kilku poziomów został pokazany na rysunkach 23 oraz 24. Umożliwia on zarządcy zawarcie nowej umowy dla wybranej nieruchomości. W nagłówku ekranu wyświetlane są podstawowe informacje o lokalu: nazwa, adres, piętro, powierzchnia pokoi. Pozwala to zweryfikować, że umowa jest tworzona dla odpowiedniej nieruchomości. Formularz jest podzielony na dwie zakładki: „Najemca i dane najmu” oraz „Opłaty”.

Ekran kreatora najmu zawiera dwie sekcje. Sekcja „Informacje dotyczące najemcy” posiada dane kontaktowe. Dostępny jest checkbox „Najem na firmę” - jego zaznaczenie rozszerza formularz o pole identyfikatora podatkowego, umożliwiając rejestrację najmu w kontekście podmiotu gospodarczego. Sekcja „Informacje dotyczące najmu” pozwalają na uzupełnienie danych odnośnie do najmu, gdzie możemy wpisać datę rozpoczęcia, datę zakończenia, typ najmu oraz uwagi.

Druga zakładka „Opłaty” umożliwia zdefiniowanie warunków finansowych umowy. Zarządca wprowadza wysokość miesięcznego czynszu, kwotę kaucji oraz dzień miesiąca, w którym płatność jest wymagana. Po prawej stronie zakładki wyświetlane jest na bieżąco aktualizowane podsumowanie umowy, zawierające typ najmu, daty obowiązywania, dane najemcy oraz zestawienie opłat. Podsumowanie pozwala zarządcy zweryfikować wszystkie wprowadzone dane przed końcowym zatwierdzeniem.

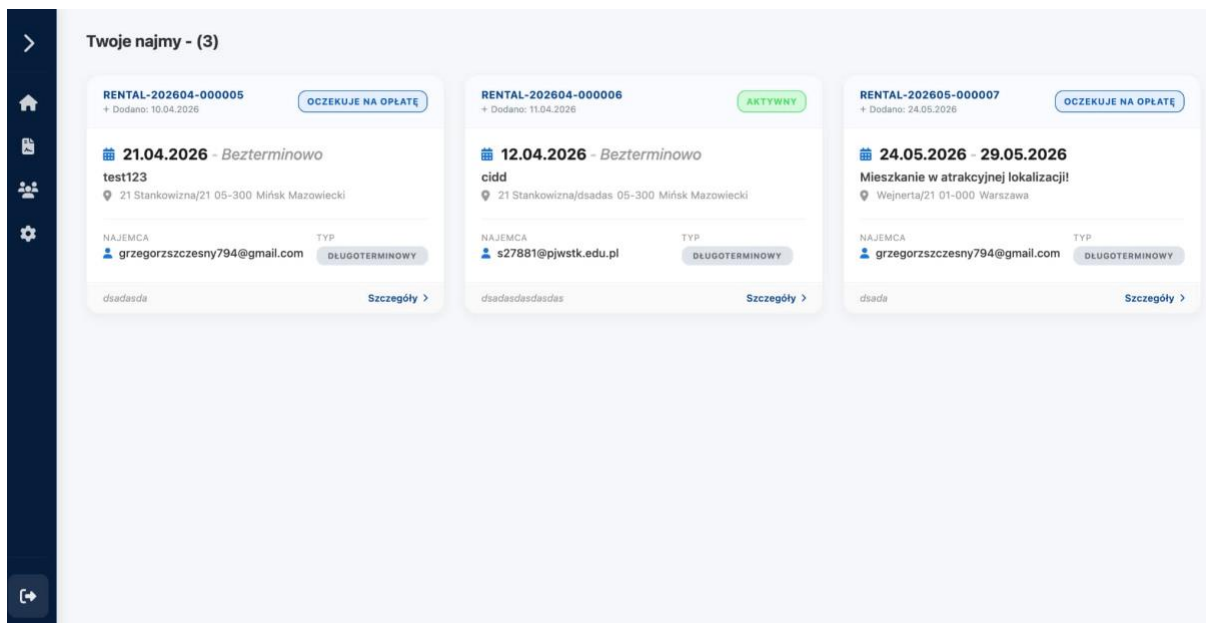
7.1.6 Ekran administratora



Rysunek 25: Ekran administratora

Ekran administratora przedstawiony na rysunku 25 jest dostępny wyłącznie dla użytkowników z rolą *LANDLORD_ADMIN_ROLE*. Przedstawia listę wszystkich kont pracowniczych przypisanych do organizacji wraz z ich aktualnymi rolami. Administrator może tworzyć nowe konta, edytować przypisane role. Nowo dodany pracownik otrzymuje e-mail z linkiem do ustawienia hasła do konta.

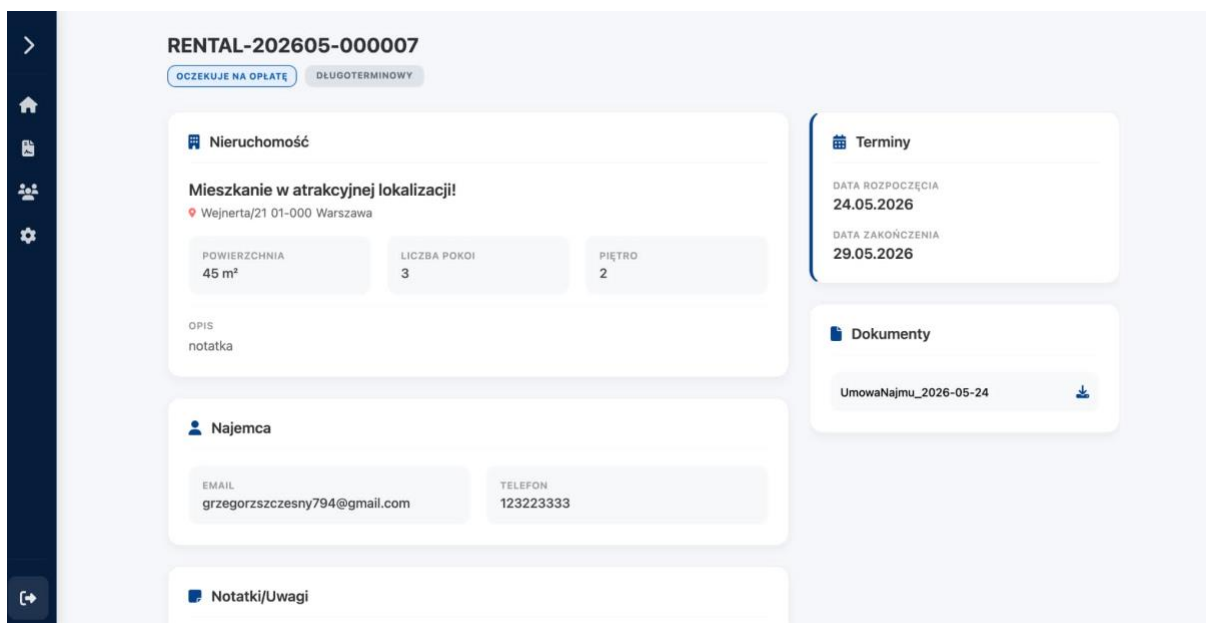
7.1.7 Ekran listy najmów



Rysunek 26: Ekran prezentujący listę przypisanych najmów do zalogowanego zarządcy

Rysunek 26 prezentuje tabelaryczną listę wszystkich najmów danego zarządcy. Każdy wiersz zawiera numer umowy najmu, daty obowiązywania umowy, dane najemcy, nazwę lokalu oraz aktualny status. Całość zamknięta na responsywnym ekranie i wyświetla kafelkowo.

7.1.8 Ekran szczegółów najmu

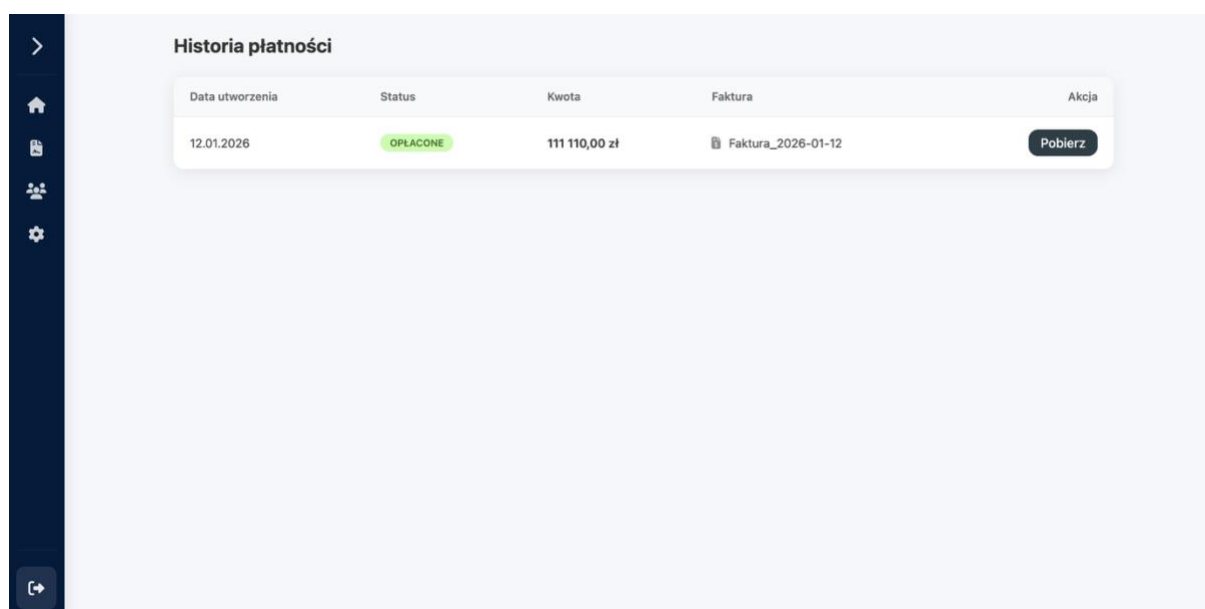


Rysunek 27: Ekran szczegółów najmu

Rysunek 27 prezentuje ekran, na którym wyświetlone są pełne informacje dotyczące wybranej umowy najmu: dane najemcy, parametry lokalu, warunki finansowe oraz harmonogram płatności z historią rozliczeń. Zarządca może z tego poziomu przeglądać wygenerowane dokumenty powiązane z umową, móc zobaczyć obecne należności, które nie zostały opłacone przez najemcę oraz śledzić i aktualny status całego procesu najmu oraz procesować przez poszczególne etapy. Dodatkowo z tego ekranu zarządca jest w stanie przenieść się do listy wszystkich opłat naliczonych oraz opłaconych od początku najmu.

7.1.9 Ekran opłat

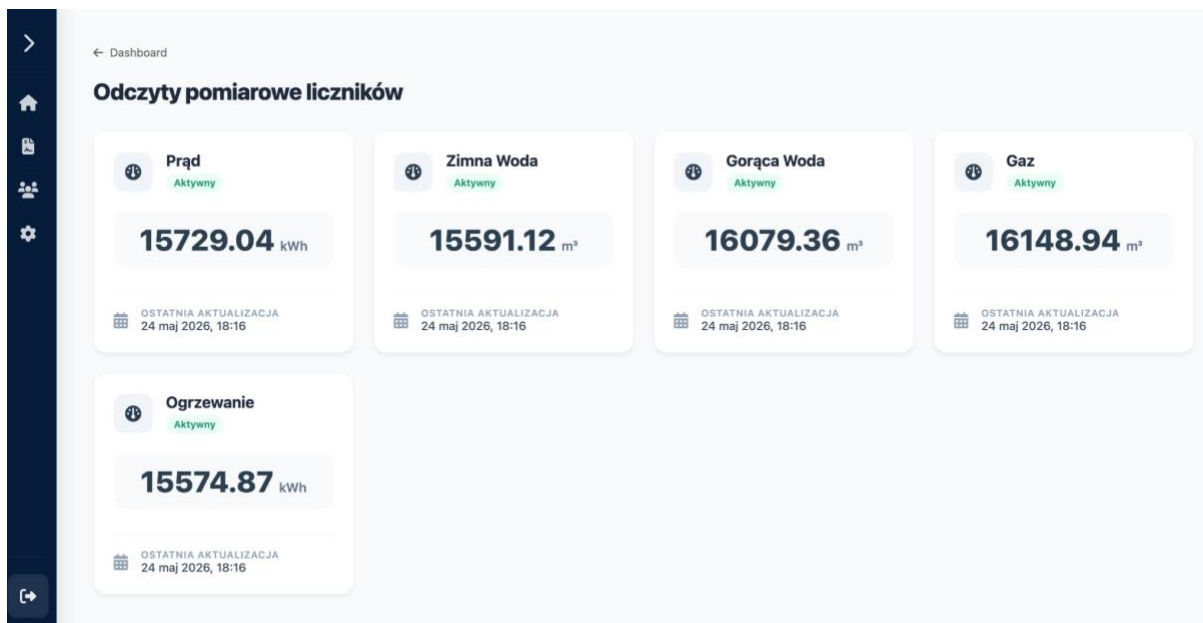
Ekran pokazany na rysunku 28 agreguje wszystkie należności w ramach tej jednej konkretnej umowy najmu. Lista opłat zawiera informacje o tytule, kwocie, terminie płatności oraz statusie każdej płatności. Widok umożliwia szybki przegląd zaległości i rozliczeń bez konieczności wchodzenia w szczegóły poszczególnych najmów. Dodatkowo zarządca ma możliwość pobrania z tego panelu utworzonego dokumentu faktury konkretnej należności, który jest generowany podczas każdej opłaty przez najemcę.



Data utworzenia	Status	Kwota	Faktura	Akcja
12.01.2026	OPLACONE	111 110,00 zł	Faktura_2026-01-12	Pobierz

Rysunek 28: Ekran wszystkich płatności za najem

7.1.10 Ekran liczników/mierników



Rysunek 29: Ekran mierników/liczników pokazujących aktualny stan w nieruchomości

Rysunek 29 pokazuje ekran zainstalowanych liczników w mieszkaniu. Służy on do przeglądania danych z ich zewnętrznych mikroprocesorów m.in. (prąd, woda, gaz) dla poszczególnych lokali. Zarządca jest w stanie odczytać jakim typem jest dany licznik, ile poszczególnych wartości jest już naliczonych od czasu instalacji oraz datę ostatniego odczytu. Obecna implementacja tego rozwiązania działa na zasadzie zamockowanych danych, które są spójnie obliczane co 30 minut. Natomiast w prawdziwej implementacji takie dane zaczytywane byłyby z prawdziwych liczników w mieszkaniach poprzez wystawione przez nie API lub API dostawcy konkretnego modelu. Dane pobierane byłyby w formie *PUSH* lub *PULL* w zależności od typu licznika i jego szyny danych jaką posiada wewnątrz i jaką wystawia na zewnątrz.

7.1.11 Ekran dodawania/modyfikacji nieruchomości

Nieruchomość

Informacje podstawowe

Wyświetlana nazwa *
ładne mieszkanie

Typ *
Apartament

Status *
Dostępne

Numer katastralny *
KW-123456

Notatki
Notatki

Informacje techniczne

Powierzchnia *
1.000 m²

Ilość pokoi *
0

Numer piętra *
0

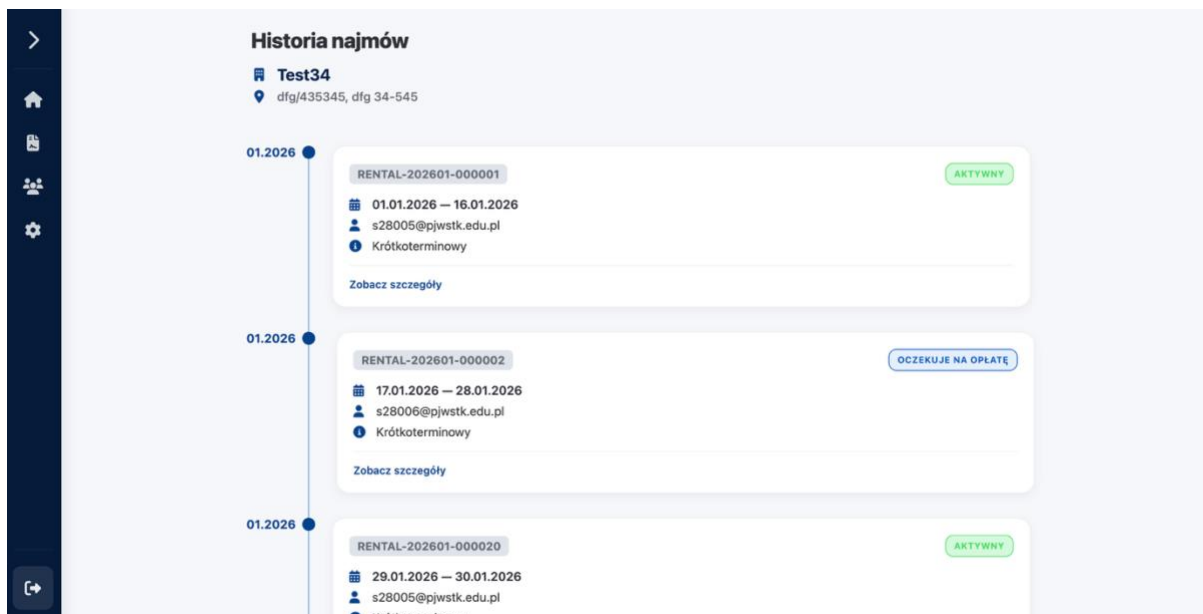
Anuluj Zapisz

Rysunek 30: Ekran umożliwiający dodanie lub aktualizację nieruchomości

Rysunek 30 przedstawia formularz tworzenia nowego lub edycji istniejącego lokalu. Formularz zawiera pola opisowe (nazwa, opis, typ, powierzchnia, liczba pokoi, piętro, numer katastralny) oraz interaktywną mapę do wskazywania lokalizacji, kliknięcie punktu na mapie automatycznie uzupełnia współrzędne geograficzne. Możliwe jest przesłanie zdjęć lokalu.

7.1.12 Ekran historii najmów dla nieruchomości

Dla każdej z nieruchomości dostępny jest podgląd wszystkich najmów, w których dana nieruchomość była elementem umowy. Ekran pokazuje linię czasu, gdzie przy konkretnej dacie widnieje kafelek z konkretnym najmem oraz informacjami o statusie, obowiązującym okresie, typie najmu oraz danym mailu, który należał do najemcy. Po kliknięciu w dany kafelek z najmem, zostajemy przesłani do szczegółów tego najmu skąd możemy zobaczyć konkretne informacje, pobrać dokumenty oraz podejrzeć historyczne należności.



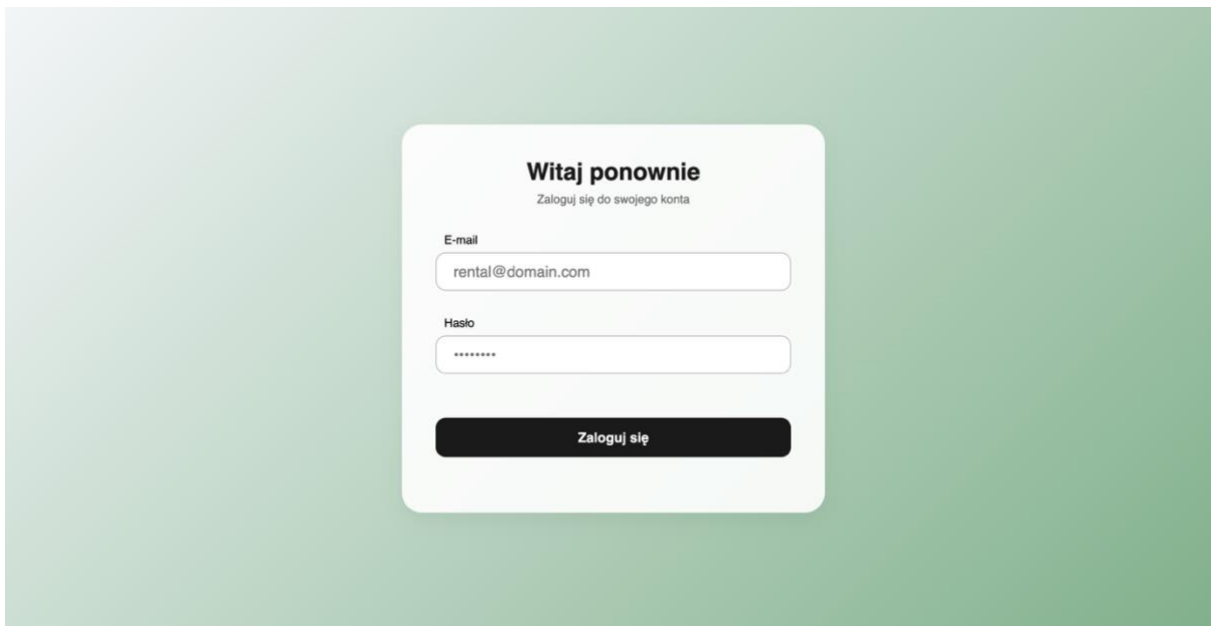
Rysunek 31: Ekran pokazujący historię najmów dla konkretnej nieruchomości

7.2. Aplikacja webowa najemcy

Następną domeną istniejącą w systemie jest domena najemcy. Dla tej konkretnej grupy odbiorców system również posiada ekrany, na których użytkownik jest w stanie podejrzeć wszystkie swoje najmy historyczne oraz aktualne, wejść w szczegóły oraz móc operować na dokumentach, śledzić statusy umowy oraz móc opłacać czynsze, kaucje jak i inne pojawiające się w trakcie najmu należności.

7.2.1 Ekran logowania

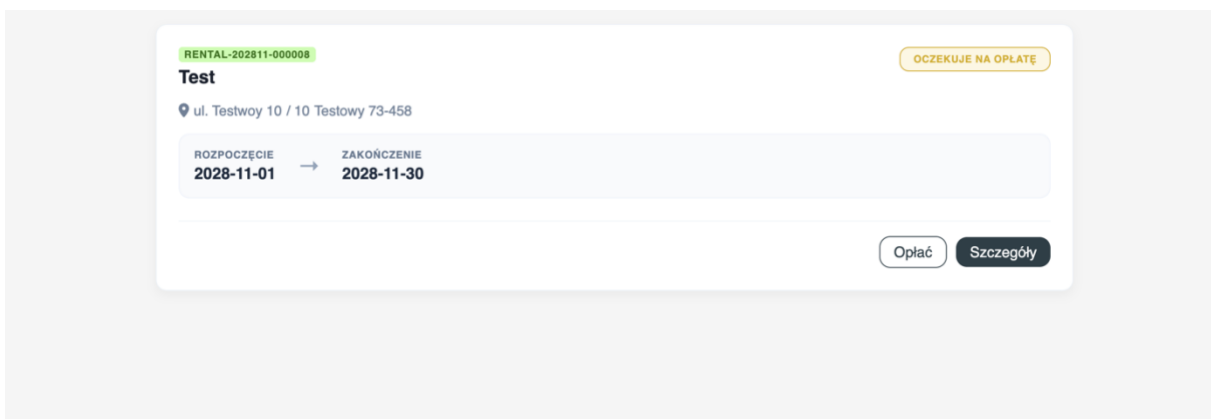
Ekran logowania przedstawiony na rysunku 32 pojawia się najemcy podczas wejścia bezpośrednio na pusty URL, tak jak to jest w przypadku panelu zarządcy. Tutaj użytkownik ma możliwość jedynie zalogowania się do swojego konta. Rejestracja najemcy bezpośrednio przez niego nie jest możliwa w systemie. Takie konto tworzone jest podczas pierwszego utworzenie najmu na dany konkretny email, po czym najemca aktywuje swoje konto i może z niego spokojnie korzystać.



Rysunek 32: Ekran pozwalający na zalogowanie się do panelu najemcy

7.2.2 Ekran listy najmów

Najemca tak samo jak zarządca ma możliwość podglądania swoich historycznych oraz aktualnych najmów. Na liście widoczne są podstawowo informacje takie jak okres obowiązywania najmu oraz m.in. nazwa i adres lokalu, który jest elementem najmu. Z tego panelu najemcy może też bezpośrednio przejść do opłaty za najem, jeśli dany najem posiada aktualne należności.

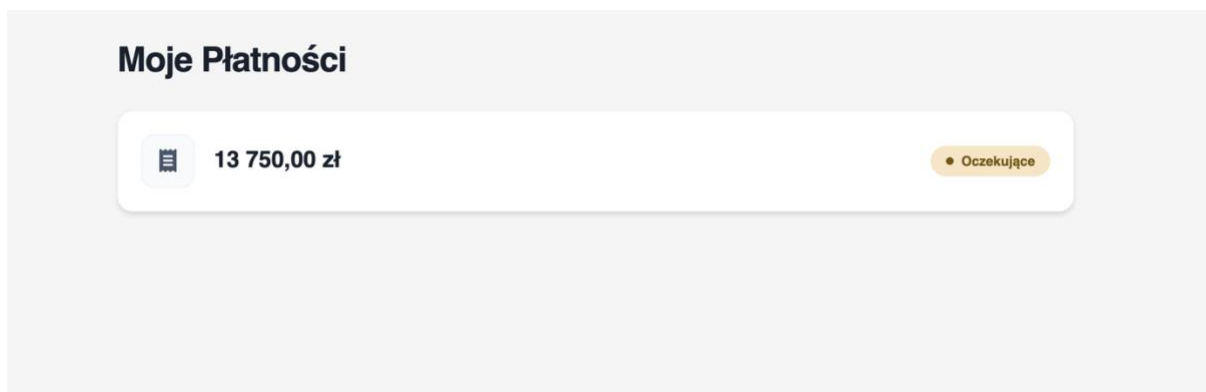


Rysunek 33: Ekran wyświetlający najmy, w których najemca bierze udział

7.2.3 Ekran listy należności

Ekran listy należności ukazany na rysunku 34 daje najemcy możliwość podglądu jego aktualnych naliczonych należności jak i również tych które zostały opłacone bądź są zaległe. Z tego miejsca najemca może pobrać fakturę każdej z opłaty głównej, w którą wliczały się

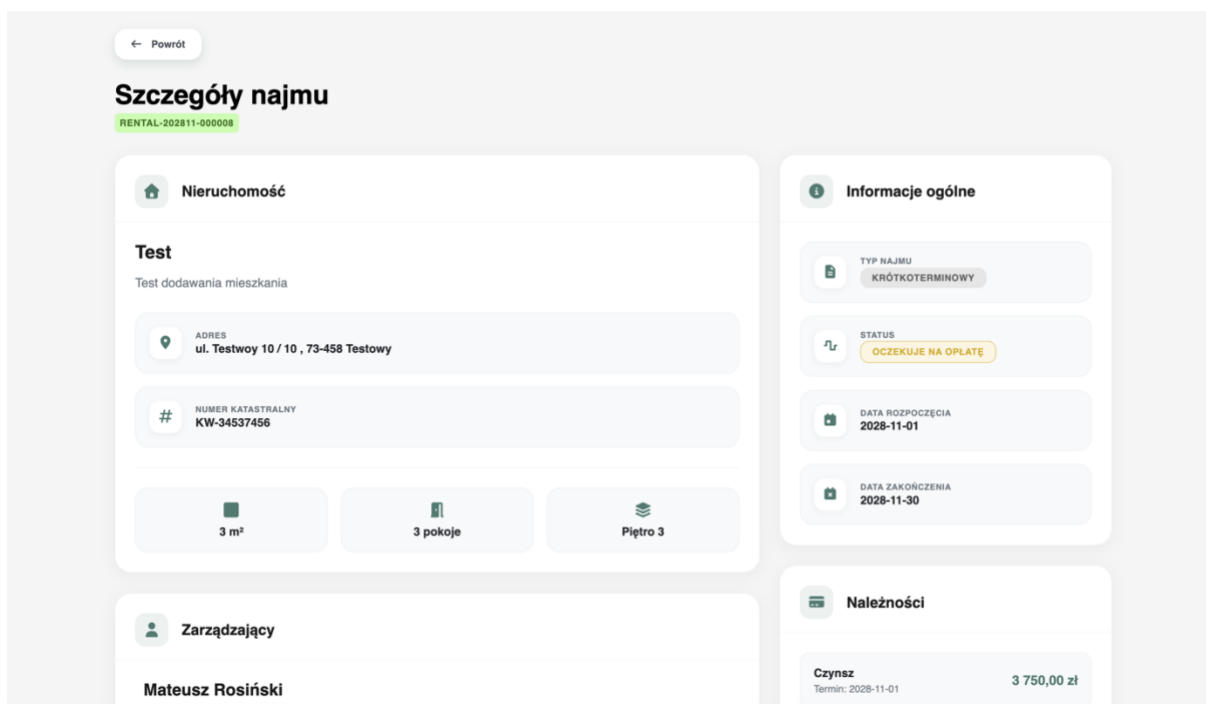
poszczególne elementy opłat. Dodatkowo z tego ekranu jesteśmy też w stanie bezpośrednio przejść na ekran płatności, żeby móc opłacić aktualne należności za najem w systemie.



Rysunek 34: Ekran listy aktywnych oraz opłaconych należności w panelu najemcy

7.2.4 Ekran szczegółów najmu

Rysunek 35 przedstawia ekran ze szczegółami konkretnego najmu, na który możemy wejść bezpośrednio z listy najmów po kliknięciu w wybrany element. Tutaj jesteśmy w stanie zobaczyć bardziej szczegółowe dane najmu jak i również dokumenty, które są przypisane pod ten konkretny najem. Dodatkowo mamy możliwość podglądu aktualnych oraz zaległych należności, które nie zostały opłacone i przejść od razu do ekranu płatności, żeby je opłacić bądź wejść do listy płatności, żeby przeglądać wszystkie swoje płatności jakie zostały naliczone za najem od początku jego zawarcia



Rysunek 35: Ekran szczegółów najmu na panelu najemcy

7.2.5 Ekran płatności

Ekran przedstawiony na rysunku 36, umożliwia bezpośrednio możliwość zapłaty za czynsz, kaucję oraz inne elementy najmu. Przedstawia poszczególne elementy za które naliczona została opłata oraz ich sumę. Ze względu na implementację jaką jest mockowanie płatności w obecnej formie, płatność następuje od razu bez wyboru żadnych metod płatności ani też integracji z zewnętrznymi bramkami płatności.

Rysunek 36: Ekran płatności na panelu najemcy

7.3. Widoki współdzielone pomiędzy aplikacjami frontendowymi

W systemie, oprócz widoków dedykowanych dla poszczególnych aplikacji frontendowych, istnieją również widoki współdzielone. Są to ekrany, które posiadają identyczny układ, wygląd oraz zachowanie w obu aplikacjach. Ich głównym celem jest zapewnienie spójności interfejsu użytkownika niezależnie od tego, z której aplikacji korzysta użytkownik.

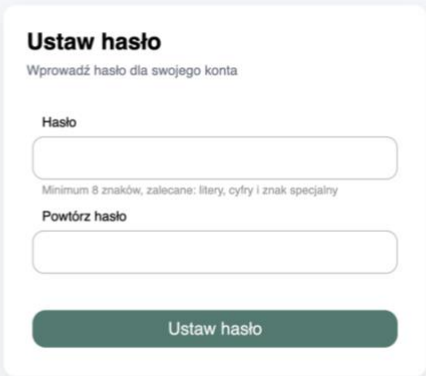
Widoki współdzielone zostały zaprojektowane w taki sposób, aby minimalizować różnice pomiędzy aplikacjami oraz ułatwiać utrzymanie systemu. Dzięki ich zastosowaniu możliwe jest ponowne wykorzystanie tych samych komponentów oraz logiki prezentacyjnej, co ogranicza duplikację kodu i zmniejsza ryzyko niespójności w działaniu interfejsu.

Zastosowanie wspólnych widoków wpływa również na łatwiejszy rozwój systemu. Wprowadzanie zmian w interfejsie odbywa się w jednym miejscu, co automatycznie przekłada się na obie aplikacje. Takie podejście zwiększa spójność wizualną oraz redukuje nakład pracy związany z utrzymaniem i rozbudową frontendów.

7.3.1 Ekran ustawienia hasła użytkownika

Jednym z widoków współdzielonych jest ekran ustawienia hasła użytkownika. Jest to widok prezentowany nowo utworzonemu użytkownikowi po przeprowadzeniu wstępnej weryfikacji konta, którego celem jest umożliwienie zdefiniowania własnego hasła dostępowego do systemu. Na tym etapie użytkownik proszony jest o dwukrotne wprowadzenie hasła w formularzu, co pozwala zminimalizować ryzyko błędów wynikających z literówek lub niepoprawnego wpisania. Po kliknięciu przycisku „Ustaw hasło” użytkownik, jest automatycznie logowany, uzyskuje systemowy token autoryzacyjny i zostaje przekierowywany do listy swoich aktualnych najmów.

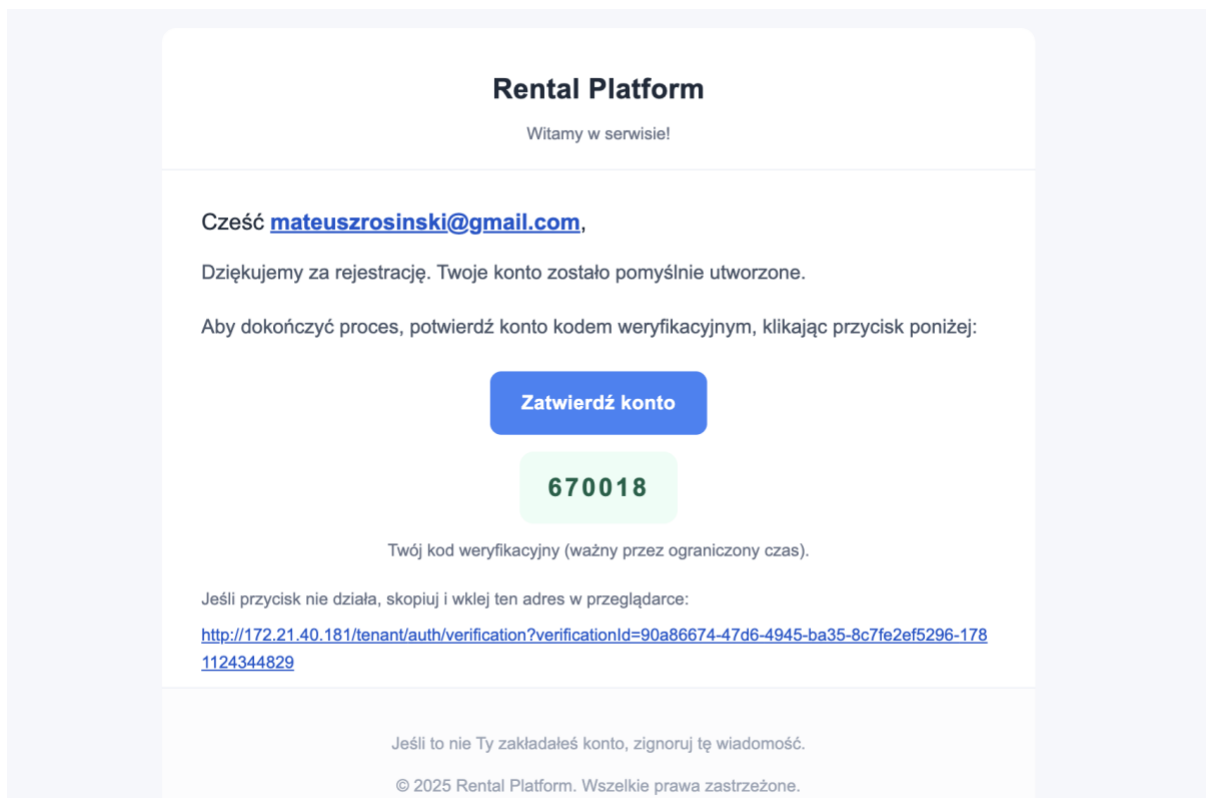
Ekran ten, który przedstawiony jest na rysunku 37 stanowi element procesu inicjalizacji konta użytkownika i jest kluczowym krokiem umożliwiającym pełne korzystanie z aplikacji. Po poprawnym zakończeniu procesu weryfikacji użytkownik zostaje przekierowany do tego widoku, gdzie ma możliwość ustalenia nowego hasła zgodnego z wymaganiami bezpieczeństwa obowiązującymi w systemie.



Rysunek 37: Ekran ustawienia hasła do konta

7.3.2 Ekran weryfikacji użytkownika po kodzie OTP

Po utworzeniu konta przez zarządcę, na dany mail podawany podczas jego tworzenia, który pokazany jest na rysunku 38 zostaje wysłana informacja z możliwością weryfikacji. Ta informacja zawiera link do strony z aktywacją konta oraz kod OTP, który należy wpisać w komponencie formularza.

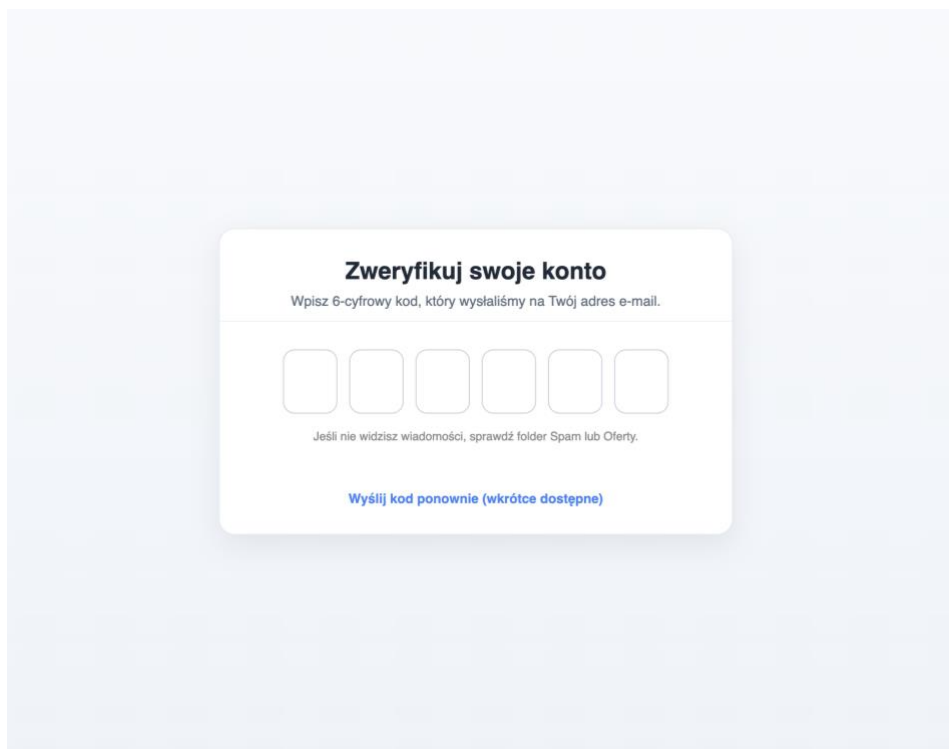


Rysunek 38: Email z kodem OTP do weryfikacji konta

Dana strona weryfikacji konta zawiera odpowiednie pole, w które należy wpisać bądź wkleić podany kod OTP. System przewiduje różne scenariusze, które mogą na niej wystąpić. Scenariuszami jakie mogą się tutaj pojawić są:

- brak kodu OTP w bazie
- przekroczenie limitu wpisania kodu do systemu
- brak ważności kodu

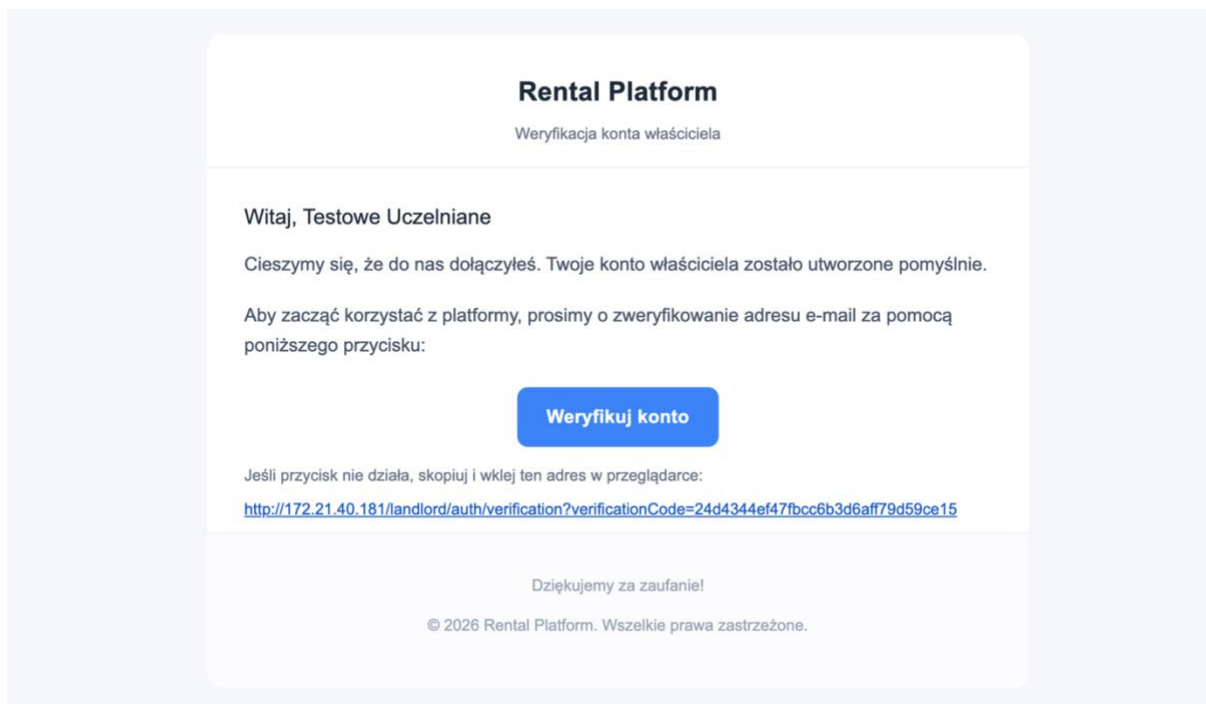
Jeśli użytkownik poprawnie się zweryfikuje jest przekierowywany na stronę, gdzie może wpisać swoje nowe hasło do konta i je odblokować.



Rysunek 39: Ekran pozwalający wpisać kod OTP

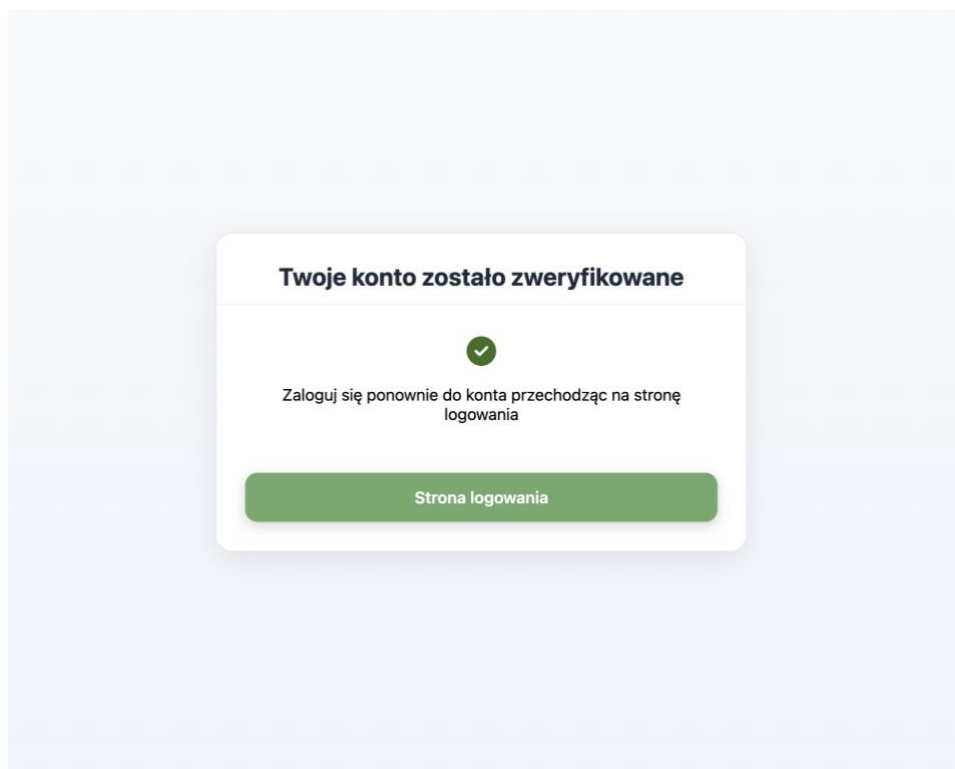
7.3.3 Ekran weryfikacji użytkownika po tokenie

Rysunek 40 przedstawia ekran weryfikacji użytkownika po tokenie stanowi element procesu aktywacji konta w systemie dla zarządców, którzy tworzą główne konto. Użytkownik trafia na ten widok poprzez link przesłany na adres e-mail, zawierający unikalny token identyfikacyjny przekazywany jako parametr w adresie URL.



Rysunek 40: Email przychodzący z kodem weryfikacyjnym

Celem tego ekranu jest potwierdzenie poprawności adresu e-mail oraz aktywacja konta użytkownika w systemie. Po wejściu w link system dokonuje weryfikacji przekazanego tokenu, sprawdzając jego ważność oraz przypisanie do konkretnego użytkownika. W przypadku poprawnego tokenu konto zostaje aktywowane, co umożliwia dalsze korzystanie z systemu. W sytuacji, gdy token jest nieprawidłowy lub wygaś, użytkownik otrzymuje odpowiedni komunikat informujący o niepowodzeniu weryfikacji.



Rysunek 41: Ekran poprawnej weryfikacji po kodzie weryfikacyjnym

8. Testy

W celu weryfikacji poprawności działania systemu przeprowadzono testy integracyjne. Ich głównym celem było sprawdzenie współpracy pomiędzy poszczególnymi elementami aplikacji, w szczególności komunikacji pomiędzy mikroservisami i ich usługami oraz poprawności przepływu danych w kluczowych procesach biznesowych.

Testy obejmowały najważniejsze funkcjonalności systemu, takie jak proces modułów związanych mechanizmami planowania zadań, poprawności danych użytkowników oraz kolejek związanych z płatnościami. Szczególną uwagę zwrócono na poprawność działania całych ścieżek użytkownika, a nie pojedynczych komponentów aplikacji.

W ramach testów sprawdzano m.in. poprawność odpowiedzi API, reakcję systemu na dane niepoprawne lub niepełne oraz zgodność działania aplikacji z założeniami funkcjonalnymi. Weryfikowano również obsługę sytuacji wyjątkowych oraz sytuacji krytycznych.

Testy integracyjne zostały przeprowadzone dla kilku kluczowych scenariuszy użycia systemu. Każdy z nich polegał na wykonaniu sekwencji operacji obejmujących współpracujące ze sobą moduły oraz porównaniu uzyskanych rezultatów z oczekiwanym zachowaniem systemu.

Przeprowadzone testy potwierdziły poprawną integrację głównych komponentów systemu oraz prawidłowe działanie kluczowych procesów użytkownika.

8.1. Testy jednostkowe

W systemie zaimplementowano testy jednostkowe. Testy jednostkowe stanowią podstawowy poziom testowania oprogramowania, którego celem jest weryfikacja poprawności działania najmniejszych, izolowanych elementów systemu, takich jak pojedyncze funkcje lub metody.

Ich istotą jest zapewnienie, że poszczególne fragmenty logiki aplikacji działają zgodnie z założeniami projektowymi, niezależnie od pozostałych komponentów systemu. Dzięki izolacji testowanych elementów możliwe jest precyzyjne wykrywanie błędów oraz ich szybka lokalizacja.

Testy jednostkowe pełnią ważną rolę w procesie wytwarzania oprogramowania, zwiększając jego niezawodność oraz ułatwiając dalszy rozwój i zamianę kodu. Stanowią również formę zabezpieczenia przed regresją, umożliwiając kontrolę poprawności działania istniejącej funkcjonalności po wprowadzeniu zmian w systemie. Testy w aplikacji oparte są o narzędzie *Mockito*, które pozwala w łatwy sposób testować dane i zależności systemu. Testy skupiają się głównie na serwisach takich jak *rental-service*, *authorization-service* oraz *notification-service*, w których to występują miejsca mocno obciążone.

8.1.1 Konfiguracja

Mockito automatycznie tworzy mocki oznaczone *@Mock* i wstrzykuje je do klasy oznaczonej *@InjectMocks*, wywołując jej konstruktor. Przykład takiej konfiguracji pokazany jest na listingu 57.

Listing 57: Konfiguracja klas mock'ujących

```
@ExtendWith(MockitoExtension.class)
class PaymentAMPQListenerTest {

    @Mock private RentalService rentalService;
    @Mock private TenantService tenantService;
    @Mock private EmailService emailService;
    @InjectMocks
    private PaymentAMPQListener paymentAMPQListener;
}
```

Istnieje również możliwość, aby klasa testowana była tworzona ręcznie w *@BeforeEach* bądź w konstruktorze jak na listingu 58, wtedy mocki tworzone są poprzez metodę *mock()*.

Listing 58: Konfiguracja mock zaimplementowana innym sposobem

```
private final RentalRepository repository = mock(RentalRepository.class);
private final EmailService emailService = mock(EmailService.class);
@BeforeEach
void setUp() {
    rentalScheduler = new RentalScheduler(repository, emailService, ...);
}
```

8.1.2 Spy - częściowe mockowanie

Metoda *spy()*, pokazana na listingu 59 tworzy obiekt oparty na prawdziwej klasie, ale pozwala nadpisać wybrane metody. Użyto go w teście aktualizacji uprawnień, kiedy to encja powinna być prawdziwym obiektem, ale jednocześnie dając jej możliwe weryfikowanie wywołań wewnątrz.

Listing 59: Operacja spy do testowania prawdziwych wywołań

```
UserEntity entity = spy(UserEntity.builder()
    .publicId("user-123")
    .authorities(new HashSet<>())
    .build());
// Przygotowanie operacji na obiektach mock
when(repository.findById("user-123")).thenReturn(Optional.of(entity));
service.update("user-123", List.of(Authority.LANDLORD_USER_ROLE));
verify(repository).save(entity);
```

9. Podsumowanie

Celem pracy inżynierskiej było zaprojektowanie oraz zaimplementowanie platformy do zarządzania najmem nieruchomości. System oparto na architekturze mikroserwisowej. W trakcie realizacji projektu powstało wiele niezależnych serwisów backendowych oraz dwie aplikacje frontendowe. Wdrożono także dedykowaną warstwę BFF dla każdego klienta. Całość działa na klastrze k3s. Proces wdrażania wspiera automatyczny potok CI/CD oparty na platformie GitLab. Niniejszy rozdział stanowi podsumowanie całości wykonanej pracy. W pierwszej kolejności przedstawiono wady i zalety przyjętych rozwiązań. Analiza ta dotyczy zarówno architektury systemu, jak i wybranych technologii. Następnie omówiono potencjalne kierunki dalszego rozwoju platformy. Mogłyby one zwiększyć jej funkcjonalność oraz wartość dla użytkowników końcowych. Rozdział zamykają wnioski końcowe. Odnoszą się one do przebiegu realizacji projektu oraz wiedzy i umiejętności zdobytych w jego trakcie.

Realizacja projektu wymagała podjęcia wielu decyzji projektowych. Ich konsekwencje są widoczne zarówno w możliwościach systemu, jak i w złożoności jego utrzymania. Świadome omówienie tych kompromisów pozwala ocenić stopień powodzenia przyjętego podejścia. Pokazuje ono, jak wybrane rozwiązania sprawdziły się w praktyce. Pozwala też określić, co można byłoby zrobić inaczej przy ponownej realizacji podobnego projektu.

9.1. Wady i zalety rozwiązania

Każde rozwiązanie techniczne wiąże się z określonymi kompromisami. Architektura mikroserwisowa przynosi duże korzyści w zakresie skalowalności i niezależności wdrożeń. Jednocześnie wprowadza złożoność, której nie ma w prostszych rozwiązaniach. Poniżej przedstawiono najważniejsze wady i zalety przyjętych rozwiązań, oceniane z perspektywy możliwości systemu oraz kosztów jego utrzymania.

Wady rozwiązania

- Ograniczona funkcjonalność w porównaniu z produktami komercyjnymi - brak aplikacji mobilnych, modułów raportowania finansowego, wbudowanej komunikacji między stronami najmu oraz integracji z portalami ogłoszeniowymi, które w systemach takich jak Buildium stanowią standard,
- Złożoność utrzymania infrastruktury - zarządzanie wieloma repozytoriami, obrazami Docker, sekretami *Kubernetes* i konfiguracją *Traefik* generuje znacznie większy narzut operacyjny niż odpowiednik w architekturze monolitycznej,
- Trudniejszy proces debugowania - śledzenie błędów w operacjach obejmujących wiele serwisów (np. inicjalizacja najmu) wymaga jednoczesnej analizy logów z kilku niezależnych komponentów, co wydłuża diagnozę problemów,
- Złożone środowisko lokalne - uruchomienie całej aplikacji wymaga jednoczesnego startu serwisów Java, serwisów C#, dwóch aplikacji Angular, bazy PostgreSQL oraz brokera RabbitMQ. *Docker Compose* ogranicza ten problem, jednak konfiguracja pozostaje bardziej wymagająca niż w monolicie,
- Narzut sieciowy przy operacjach wieloserwisowych - operacje wymagające danych z wielu serwisów (np. wyświetlenie szczegółów najmu) generują wiele oddzielnych zapytań HTTP, co zwiększa opóźnienia i komplikuje obsługę błędów.

Zalety rozwiązania

- Pełna kontrola nad danymi i infrastrukturą - dane najemców i umów przechowywane są wyłącznie na własnej infrastrukturze, bez udziału zewnętrznych dostawców, co zapewnia zgodność z wymogami RODO,
- Niezależny *pipeline* CI/CD dla każdego serwisu - zmiana w dowolnym serwisie nie wymaga restartu pozostałych komponentów, co skraca czas wdrożenia i minimalizuje ryzyko regresji,
- Swoboda technologiczna - architektura mikroservisowa umożliwia dobór technologii odpowiedniej do wymagań danej domeny: logika biznesowa najmu i autoryzacji zaimplementowana jest w Javie ze Spring Boot, natomiast generowanie dokumentów PDF realizuje serwis napisany w C# z wykorzystaniem biblioteki PuppeteerSharp - rozwiązanie niemożliwe do zastosowania w architekturze monolitycznej,
- Odporność na awarie dzięki komunikacji asynchronicznej - zastosowanie RabbitMQ jako brokera wiadomości zapewnia, że powiadomienia e-mail nie są tracone w przypadku chwilowej niedostępności serwisu - wiadomości oczekują w kolejce i zostają dostarczone po jego ponownym uruchomieniu,
- Zarządzanie schematem bazy danych przez Liquibase - pełna historia migracji przechowywana jest w repozytorium kodu. Wdrożenie nowej wersji serwisu automatycznie aktualizuje strukturę tabel, a w razie potrzeby możliwe jest cofnięcie zmian przez mechanizm rollback,
- Izolacja frontendu od logiki wewnętrznej dzięki wzorcowi BFF - serwisy landlord-backend i tenant-backend pełnią rolę fasady ukrywającej architekturę wewnętrzną przed aplikacjami Angular. Każdy BFF eksponuje wyłącznie okrojone DTO dedykowane konkretnemu widokowi, zwalniając serwisy domenowe z konieczności samodzielnej weryfikacji tokenów JWT,
- Kompletny cykl życia najmu - platforma obsługuje cały proces: od rejestracji wynajmującego i weryfikacji konta firmowego przez GUS, przez dodanie nieruchomości i zawarcie umowy z najemcą, aż po automatyczne powiadomienia e-mail o zbliżającym się zakończeniu umowy, generowanie dokumentu PDF oraz rozliczenie płatności.

9.2. Potencjalne kierunki rozwoju

Integracja z bramką płatniczą

Kolejnym naturalnym krokiem byłoby podłączenie systemu do bramki płatniczej np. Stripe lub Przelewy24. Pozwoliłoby to najemcom opłacać czynsz bezpośrednio w poprzez swoje konta bankowe. Obecny moduł płatności rejestruje opłaty i ich statusy, ale faktycznie rozliczenia są mockowane przez system. Integracja zamknęłaby pełny cykl finansowy wewnątrz platformy i wyeliminowała moduł mockowania.

Komunikacja wewnętrzna

System nie oferuje żadnego kanału komunikacji między wynajmującym, a najemcą. Dodanie modułu chatów z możliwością pisania wiadomości lub systemu zgłoszeń usterek pozwoliłoby obsłużyć topowe scenariusze, raportowanie awarii przez najemcę, potwierdzenie

stanu technicznego lokalu czy ustalenie dotyczące rozwiązania umowy, bez konieczności wychodzenia poza platformę.

Aplikacja mobilna

Obie aplikacje frontendowe są platformami internetowymi. Stworzenie aplikacji mobilnej, zwiększyłoby dostępność platformy, co przyczyniłoby się do zwiększenia realnych zysków osób prowadzących swoje firmy poprzez naszą aplikację. Istniejące BFF (tenant-backend, landlord-backend) mogłyby obsługiwać aplikację mobilną bez zmian w warstwie serwerowej, ponieważ już teraz pełnią rolę fasady, która izoluje frontend od wewnętrznych serwisów.

Moduł raportowania finansowego

Właściciele nieruchomości potrzebują zestawień przychodów z najmu na potrzeby rozliczeń podatkowych. Rozbudowa modułu płatności o generowanie raportów miesięcznych i rocznych, eksport do PDF lub CSV oraz obsługę różnych stawek VAT byłaby wartością dodaną szczególnie dla wynajmujących zarządzających większym portfelem nieruchomości.

Logowanie i monitoring

Wraz z rosnącą liczbą serwisów debugowanie przepływu zapytań przez kilka mikroservisów staje się coraz bardziej skomplikowane. Wdrożenie *OpenTelemetry* z eksportem do *Jaeger* lub *Grafana Tempo* pozwoliłoby śledzić czas życia pojedynczego żądania od BFF przez serwisy domenowe aż do bazy danych, co skróciłoby czas diagnozy błędów produkcyjnych.

Integracja z zewnętrznymi portalami ogłoszeniami

Automatyczne publikowanie ogłoszeń o dostępnych nieruchomościach na portalach takich jak OLX czy Otodom po zakończeniu najmu skróciłoby czas, w którym konkretny lokal jest wolny i dostępny dla nowego najemcy. Wymagałoby to rozbudowy modułu nieruchomości o obsługę galerii zdjęć w kontekście ogłoszenia i integracji z API zewnętrznych serwisów oraz odpowiednie zabezpieczenia.

9.3. Wnioski

Projekt pozwolił zrealizować kompletną platformę do zarządzania najmem nieruchomości. System obsługuje pełny cykl życia najmu. Rozpoczyna się on od rejestracji konta wynajmującego i weryfikacji danych firmowych przez GUS. Następnie wynajmujący dodaje nieruchomość i zawiera umowę z najemcą. Po opłaceniu pierwszej płatności najem staje się aktywny. System automatycznie przypomina obu stronom o zbliżającym się terminie zakończenia umowy. Po jego upływie najem jest zamykany, generowany jest dokument rozliczeniowy i nieruchomość wraca do puli dostępnych. Każdy z tych etapów jest obsługiwany przez dedykowany komponent systemu.

Wybór architektury mikroservisowej okazał się trafny. Podział na niezależne serwisy umożliwił zastosowanie różnych technologii tam, gdzie miało to uzasadnienie. Generowanie dokumentów PDF zostało zrealizowane w C# z biblioteką PuppeteerSharp. Logika biznesowa

najmu i autoryzacji powstała w Javie ze Spring Boot. W architekturze monolitycznej taki podział nie byłby możliwy.

Automatyzacja wdrożeń przez GitLab CI/CD usprawniła proces wytwarzania oprogramowania. Każda zmiana w kodzie uruchamia potok budowania i wdrożenia. Skraca to czas od modyfikacji do jej pojawienia się w środowisku produkcyjnym. Praca nad projektem pozwoliła zdobyć praktyczne doświadczenie w wielu obszarach. Projektowanie systemu rozproszonego, konfiguracja klastra Kubernetes, obsługa kolejek RabbitMQ oraz implementacja serwera autoryzacyjnego OAuth2 to zagadnienia, których nie sposób w pełni zrozumieć bez bezpośredniej pracy z nimi. Zdobyte umiejętności stanowią wartościowe uzupełnienie wiedzy teoretycznej uzyskanej w trakcie studiów.

Prace cytowane

- [1] Narodowy Bank Polski, „Raport o sytuacji na rynku nieruchomości mieszkaniowych i komercyjnych w Polsce w 2024 roku, s. 55-56, 60-62, 70-72,” 2025. [Online]. Available: <https://nbp.pl/wp-content/uploads/2025/10/Raport-o-sytuacji-na-ryнку-nieruchomosci-2024-RAPORT.pdf>. [Data uzyskania dostępu: 03 06 2026].
- [2] Bankier.pl, „Ceny transakcyjne mieszkań – raport / analiza rynku nieruchomości,” [Online]. Available: <https://www.bankier.pl>. [Data uzyskania dostępu: 03 06 2026].
- [3] Booking, [Online]. Available: <https://booking.com>. [Data uzyskania dostępu: 15 05 2026].
- [4] Airbnb, [Online]. Available: <https://airbnb.pl>. [Data uzyskania dostępu: 15 05 2026].
- [5] Lanfolk, [Online]. Available: <https://landfolk.com>. [Data uzyskania dostępu: 15 05 2026].
- [6] Modena Invest Sp. z o.o., „Rentio,” [Online]. Available: <https://rentio.com.pl/landing>. [Data uzyskania dostępu: 03 06 2026].
- [7] Apartemo, „Apartemo,” [Online]. Available: <https://apartemo.com/pl-PL/>. [Data uzyskania dostępu: 03 06 2026].
- [8] FlatFlow, „FlatFlow,” [Online]. Available: <https://flatflow.pl/>. [Data uzyskania dostępu: 03 06 2026].
- [9] RentFLow, „RentFlow,” [Online]. Available: <https://www.rentflow.io/pl/>. [Data uzyskania dostępu: 03 06 2026].
- [10] RealPage, „Buildium,” [Online]. Available: <https://www.buildium.com/>. [Data uzyskania dostępu: 03 06 2026].
- [11] J. S. J, The Art of Doing Twice the Work in Half the Time.
- [12] Scrum, „Oficjalny opis frameworku od Jeffa Sutherlanda i Kena Schwabera,” [Online]. Available: <https://scrumguides.org>. [Data uzyskania dostępu: 03 06 2026].
- [13] Atlassian, „dokumentacja i przewodniki,” [Online]. Available: <https://www.atlassian.com/software/jira>. [Data uzyskania dostępu: 03 06 2026].

- [14] Google Engineering Practices, „Code Review Guide,” [Online]. Available: <https://google.github.io/eng-practices/review/>. [Data uzyskania dostępu: 03 06 2026].
- [15] Spring Security, [Online]. Available: <https://spring.io/projects/spring-security>. [Data uzyskania dostępu: 17 05 2026].
- [16] OWASP Foundation, [Online]. Available: <https://owasp.org/www-project-top-ten/>. [Data uzyskania dostępu: 17 05 2026].
- [17] Eclipse Foundation, „Jakarta Bean Validation Specification 3.1.,” [Online]. Available: <https://jakarta.ee/specifications/bean-validation/3.1/>. [Data uzyskania dostępu: 18 05 2026].
- [18] Oracle, „Java SE 21 Documentation, 2023,” [Online]. Available: <https://docs.oracle.com/en/java/javase/21/>. [Data uzyskania dostępu: 22 05 2026].
- [19] VMWare, „Spring Boot Reference Documentation,” [Online]. Available: <https://spring.io/projects/spring-boot>. [Data uzyskania dostępu: 22 05 2026].
- [20] Apache Software Foundation, „Apache Tomcat 10.1 Documentation,” [Online]. Available: <https://tomcat.apache.org/tomcat-10.1-doc>. [Data uzyskania dostępu: 22 05 2026].
- [21] VMware, „Spring Boot Actuator Reference Guide,” [Online]. Available: <https://docs.spring.io/spring-boot/reference/actuator/>. [Data uzyskania dostępu: 22 05 2026].
- [22] Microsoft, „Documentation,” [Online]. Available: <https://learn.microsoft.com/en-gb/dotnet/csharp/>. [Data uzyskania dostępu: 22 05 2026].
- [23] Microsoft, „.NET 9 Documentation,” [Online]. Available: <https://learn.microsoft.com/dotnet/>. [Data uzyskania dostępu: 22 05 2026].
- [24] Microsoft, „ASP.NET Core Documentation,” [Online]. Available: <https://learn.microsoft.com/aspnet/core/>. [Data uzyskania dostępu: 22 05 2026].
- [25] PostgreSQL Global Development Group, „PostgreSQL 17 Documentation,” [Online]. Available: <https://www.postgresql.org/docs/>. [Data uzyskania dostępu: 22 05 2026].
- [26] Google, „Angular 19 Documentation,” [Online]. Available: <https://v19.angular.dev/overview>. [Data uzyskania dostępu: 22 05 2016].

- [27] Microsoft, „TypeScript Documentation,” [Online]. Available: <https://www.typescriptlang.org/docs/>. [Data uzyskania dostępu: 22 05 2026].
- [28] VMWare, „RabbitMQ Documentation,” [Online]. Available: <https://www.rabbitmq.com/docs>. [Data uzyskania dostępu: 22 05 2026].
- [29] Leaflet, „Leaflet API Reference,” [Online]. Available: <https://leafletjs.com/>. [Data uzyskania dostępu: 24 05 2026].
- [30] RxJs, „RxJS Documentation,” [Online]. Available: <https://rxjs.dev/>. [Data uzyskania dostępu: 24 05 2026].
- [31] ReactiveX, „Reactive Extensions (Rx),” [Online]. Available: <https://reactivex.io/>. [Data uzyskania dostępu: 24 05 2026].
- [32] NgRx, „NgRx Documentation,” [Online]. Available: <https://ngrx.io/>. [Data uzyskania dostępu: 24 05 2026].
- [33] Mockito, „Mockito Documentation,” [Online]. Available: <https://site.mockito.org/>. [Data uzyskania dostępu: 24 05 2026].
- [34] Liquibase, „Liquibase Official Website,” [Online]. Available: <https://www.liquibase.com/>. [Data uzyskania dostępu: 24 05 2026].
- [35] PuppeteerSharp, „API Reference,” [Online]. Available: <https://www.puppeteerssharp.com/api/>. [Data uzyskania dostępu: 24 05 2026].
- [36] Entity Framework Core, „EF Core Documentation. Microsoft Learn,” [Online]. Available: <https://learn.microsoft.com/en-us/ef/core/>. [Data uzyskania dostępu: 24 05 2026].
- [37] Hibernate ORM, „Hibernate ORM Official Website,” [Online]. Available: <https://hibernate.org/orm/>. [Data uzyskania dostępu: 24 05 2026].
- [38] Docker, „Docker Documentation,” [Online]. Available: <https://docs.docker.com/>. [Data uzyskania dostępu: 24 05 2026].
- [39] Kubernetes, „Kubernetes Official Website,” [Online]. Available: <https://kubernetes.io/docs/home/>. [Data uzyskania dostępu: 24 05 2026].
- [40] Traefik, „Traefik Official Website,” [Online]. Available: <https://doc.traefik.io/traefik/>. [Data uzyskania dostępu: 24 05 2026].

- [41] Caffeine, „Caffeine Github Repository,” [Online]. Available: <https://github.com/ben-manes/caffeine>. [Data uzyskania dostępu: 24 05 2026].
- [42] Baeldung, „Introduction to Caffeine Cache in Java,” [Online]. Available: <https://www.baeldung.com/java-caching-caffeine>. [Data uzyskania dostępu: 24 05 2026].
- [43] Swagger, „Swagger Documentation,” [Online]. Available: <https://swagger.io/docs/>. [Data uzyskania dostępu: 24 05 2026].
- [44] Project Lombok, „Project Lombok Documentation,” [Online]. Available: <https://projectlombok.org/features/>. [Data uzyskania dostępu: 24 05 2026].
- [45] Dev Community, „Monolith vs Microservices,” [Online]. Available: <https://dev.to/alexnicolascode/monolith-vs-microservices-when-to-use-each-architecture-2j92>. [Data uzyskania dostępu: 06 06 2026].
- [46] GeekForGeeks, „Monolith vs Microservices,” [Online]. Available: <https://www.geeksforgeeks.org/software-engineering/monolithic-vs-microservices-architecture/>. [Data uzyskania dostępu: 10 06 2026].
- [47] McKinsey And Company, „Ten antipatterns that are derailing technology transformations,” [Online]. Available: <https://www.mckinsey.com/capabilities/tech-and-ai/our-insights/ten-antipatterns-that-are-derailing-technology-transformations>. [Data uzyskania dostępu: 06 06 2026].
- [48] Medium, „Monolithic Architecture And Message Bus With Apache Kafka,” [Online]. Available: <https://medium.com/%40mucagriaktas/monolithic-architecture-and-message-bus-with-apache-kafka-55e8fd5030c8>. [Data uzyskania dostępu: 10 06 2026].
- [49] GeekForGeeks, „How Web Works - Web Application Architecture,” [Online]. Available: <https://www.geeksforgeeks.org/blogs/how-web-works-web-application-architecture-for-beginners/>. [Data uzyskania dostępu: 10 06 2026].
- [50] Medium, „Pierwsze kroki w Kubernetes: klastry i pody,” [Online]. Available: <https://zanondev.medium.com/teste-d25a9dde185d>. [Data uzyskania dostępu: 10 06 2026].

- [51] Medium, „Hexagonal Architecture: a Practical Review,” [Online]. Available: <https://medium.com/%40mortezatavakoli/practical-review-of-hexagonal-architecture-97ba21fd7957>. [Data uzyskania dostępu: 10 06 2026].
- [52] Medium, „AWS Serverless Solution using S3, API Gateway, SQS, Lambda and Dynamo DB with Spring boot,” [Online]. Available: <https://medium.com/%40ahujarahul716/aws-serverless-solution-using-s3-api-gateway-sqs-lambda-and-dynamo-db-with-spring-boot-a99259d5cdb2>. [Data uzyskania dostępu: 10 06 2026].
- [53] DigitalOcean, „REST Fundamentals,” [Online]. Available: <https://restful-api.dev/rest-fundamentals/>. [Data uzyskania dostępu: 10 06 2026].
- [54] IBM, „What is REST,” [Online]. Available: <https://www.ibm.com/think/topics/rest-apis>. [Data uzyskania dostępu: 06 06 2026].
- [55] MDN, „HTTP Request Methods,” [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods>. [Data uzyskania dostępu: 06 06 2026].
- [56] Mendix, „Asynchronous vs Synchronous programming. Key Similarities and differences,” [Online]. Available: <https://www.mendix.com/blog/asynchronous-vs-synchronous-programming/>. [Data uzyskania dostępu: 06 06 2026].
- [57] Hardt, D. (Ed.), „*The OAuth 2.0 Authorization Framework*, RFC 6749, Internet Engineering Task Force (IETF), październik 2012,” [Online]. Available: <https://www.rfc-editor.org/info/rfc6749/>. [Data uzyskania dostępu: 06 06 2026].
- [58] Jones, M. B., Bradley, J., & Sakimura, N, „JSON Web Token (JWT) (RFC 7519),” [Online]. Available: <https://www.rfc-editor.org/info/rfc7519/>. [Data uzyskania dostępu: 06 06 2026].

Spis listingów

Listing 1: Podstawowa konfiguracja liquibase.....	44
Listing 2: Konfiguracja changelog	45
Listing 3: Konfiguracja budowania obrazu w rejestrze	46
Listing 4: Konfiguracja w pliku docker-compose.yml serwisu oraz jego zmiennych środowiskowych.....	47
Listing 5: Konfiguracja ingress dla aplikacji klienckiej.....	49
Listing 6: Konfiguracja Gitlab CI/CD przedstawiająca kolejne etapy automatyzacji	50
Listing 7: Konfiguracja Docker DinD dla projektu	51
Listing 8: Implementacja klienta REST oraz jego użycie.....	72
Listing 9: Emisja wewnętrznego zdarzenia po poprawnym zapisie do bazy danych	73
Listing 10: Odebranie zdarzenia i obsługa dalszej części	73
Listing 11: Implementacja klasy RabbitTemplate odpowiedzialnej za wysyłkę zdarzeń.....	73
Listing 12: Metoda nasłuchująca na zdarzenie płatności	73
Listing 13: Metoda nasłuchująca na zdarzenie związane z mailem.....	73
Listing 14: Implementacja tokenów pośredniczących w autoryzacji.....	75
Listing 15: Implementacja tworzenia certyfikatu.....	76
Listing 16: Konfiguracja podstawy OAuth 2.0	77
Listing 17: Przykład utworzenia indeksu pojedynczego na kolumnie bank_account_id.....	81
Listing 18: Implementacja rozszerzenia BTree GiSTs do bazy PostgreSQL	82
Listing 19: Mapowane encji bazodanowych na obiekty domenowe.....	83
Listing 20: Przykład implementacji interfejsu z danymi	83
Listing 21: Utworzony agregat gotowy do zaimplementowania	84
Listing 22: Implementacja serwisu domenowego	84
Listing 23: Implementacja repozytorium domenowego.....	85
Listing 24: Adnotacja @Builder tworzy obiekt przygotowany pod ten wzorec, który możemy użyć za pomocą statycznej metody builder()	89
Listing 25: Implementacja wzorca ACL podczas mapowania danych wejściowych na domenowe	90
Listing 26: Implementacja wzorca Adapter	90
Listing 27: Utworzenie klasy abstrakcyjnej <i>DocumentBuilder</i> z przekazaniem typu generycznego do implementacji wzorca dla obiektów dziedziczących po niej	91

Listing 28: Implementacja wzorca Fabryki, aby móc dostosować wychodzące obiekty z domeny w odpowiedni sposób do odpowiednich operacji kontekstowych	92
Listing 29: Przykład interfejsu Landlord oraz możliwych implementacji	94
Listing 30: Przykład interfejsu gotowego do implementacji na klasie	95
Listing 31: Abstrakcyjna klasa AbstractInputComponent dostępna do dziedziczenia przez współdzielona pola formularzy	98
Listing 32: Metody pozwalające na wyświetlenie powiadomień o konkretnym typie	99
Listing 33: Implementacja mechanizmu Overlay	100
Listing 34: Implementacja mechanizmu ComponentPortal	100
Listing 35: Zastosowanie metody filtrującej w komponencie daty z Angular Material	101
Listing 36: Implementacja konfiguracji daty	102
Listing 37: Implementacja współdzielonych struktur dostępnych dla innych serwisów	103
Listing 38: Przykład obsługi operacji na kliencie serwisu	105
Listing 39: Implementacja interceptora przekazującego token użytkownika do następnych mikroserwisów	106
Listing 40: Implementacja autoryzacji do określonych endpointów	106
Listing 41: Implementacja szukania użytkowników razem z kontekstem	107
Listing 42: Implementacja pobierania kontekstu	107
Listing 43: Implementacja klucza do podpisywania tokenów	108
Listing 44: Lista ról przypisana do głównego zarządcy	109
Listing 45: Implementacja inicjalizacji narzędzia do generowania dokumentów PuppeteerSharp.....	109
Listing 46: Implementacja metody generującej dokument PDF	110
Listing 47: Implementacja metody potrzebnej do streamowania danych z bazy	110
Listing 48: Użycie metody streamującej z bazy danych, w klauzuli try	111
Listing 49: Użycie cache do optymalizacji operacji	111
Listing 50: Implementacja wyboru odpowiedniego typu dla licznika	112
Listing 51: Implementacja zapisu nowych mockowanych wartości licznika	112
Listing 52: Przetwarzanie danych partiami, wykorzystując interfejs CompletableFuture do operacji asynchronicznych	113
Listing 53: Implementacja buforowania danych z innego mikroserwisu przy pomocy cache'owania.....	114

Listing 54: Implementacja operacji obsługi zdarzenia oraz odbioru	114
Listing 55: Konfiguracja docker-compose jednego z serwisów	116
Listing 56: Konfiguracja automatyzacji Gitlab	117
Listing 57: Konfiguracja klas mock'ujących	143
Listing 58: Konfiguracja mock zaimplementowana innym sposobem	143
Listing 59: Operacja spy do testowania prawdziwych wywołań	143

Spis rysunków

Rysunek 1: Główne okno projektu na platformie Jira, źródło: opracowanie własne	14
Rysunek 2: Zrzut ekranu pokazujący główny panel aplikacji Rentio, źródło: opracowanie własne	18
Rysunek 3: Zrzut ekranu Panelu nieruchomości na platformie Apartemo, Źródło: [7]	19
Rysunek 4: Strona główna Swagger z opisem endpointów mikroserwisu, Źródło: opracowanie własne	53
Rysunek 5: Użycie Postmana do testowania endpointów, Źródło: Opracowanie własne	54
Rysunek 6: Przykład architektury monolitycznej, źródło: [46]	59
Rysunek 7: Przykład architektury mikroserwisowej, źródło: [46]	60
Rysunek 8: Architektura monolityczna, źródło: [48]	61
Rysunek 9: Przykład architektury wielowarstwowej, źródło: [49]	62
Rysunek 10: Architektura mikroserwisowa, źródło: [50]	64
Rysunek 11: Architektura heksagonalna, źródło: [51]	65
Rysunek 12: Architektura serverless, źródło: [52]	66
Rysunek 13: Diagram architektury systemu	69
Rysunek 14: Diagram REST API, źródło: [53]	70
Rysunek 15: Przykład dostępnych metod dla protokołu HTTP	71
Rysunek 16: Ekran logowania do panelu zarządcy	120
Rysunek 17: Ekran pierwszego kroku podczas rejestracji zarządcy	121
Rysunek 18: Ekran drugiego kroku podczas rejestracji zarządcy dla konta firmowego	122
Rysunek 19: Ekran kroku drugiego podczas rejestracji zarządcy dla konta indywidualnego	123
Rysunek 20: Ekran trzeciego podczas rejestracji zarządcy	124

Rysunek 21: Panel główny w aplikacji zarządcy	125
Rysunek 22: Ekran ustawień w panelu zarządcy	126
Rysunek 23: Ekran kreatora najmu - zakładka „Najemca i dane najmu”	127
Rysunek 24: Ekran kreatora najmu - zakładka „Opłaty”	127
Rysunek 25: Ekran administratora	128
Rysunek 26: Ekran prezentujący listę przypisanych najmów do zalogowanego zarządcy....	129
Rysunek 27: Ekran szczegółów najmu.....	129
Rysunek 28: Ekran wszystkich płatności za najem.....	130
Rysunek 29: Ekran mierników/liczników pokazujących aktualny stan w nieruchomości	131
Rysunek 30: Ekran umożliwiający dodanie lub aktualizację nieruchomości	132
Rysunek 31: Ekran pokazujący historię najmów dla konkretnej nieruchomości.....	133
Rysunek 32: Ekran pozwalający na zalogowanie się do panelu najemcy.....	134
Rysunek 33: Ekran wyświetlający najmy, w których najemca bierze udział	134
Rysunek 34: Ekran listy aktywnych oraz opłaconych należności w panelu najemcy	135
Rysunek 35: Ekran szczegółów najmu na panelu najemcy.....	135
Rysunek 36: Ekran płatności na panelu najemcy	136
Rysunek 37: Ekran ustawienia hasła do konta	137
Rysunek 38: Email z kodem OTP do weryfikacji konta	138
Rysunek 39: Ekran pozwalający wpisać kod OTP.....	139
Rysunek 40: Email przychodzący z kodem weryfikacyjnym	140
Rysunek 41: Ekran poprawnej weryfikacji po kodzie weryfikacyjnym	141