

POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Maciej Alfut s27322
Kacper Gajdamowicz s27372
Michał Gołaszewski s26525
Łukasz Lendzion s27952
Wojciech Matusiewicz s27661
Piotr Minta s29460
Maciej Sobolewski s26519
Michał Stasik s27525

Wellhicle – platforma do zarządzania wypożyczalniami pojazdów w modelu SaaS

Praca inżynierska napisana pod kierunkiem:

dr. inż. Mariusza Trzaski, prof. PJATK

Warszawa, lipiec, 2026

Warszawa, dnia 21.6.2026

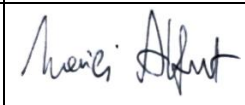
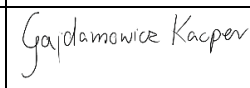
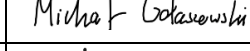
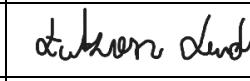
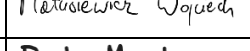
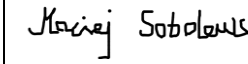
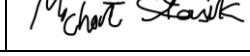
OŚWIADCZENIE WSPÓŁAUTORÓW DOTYCZĄCE WYKORZYSTANIA SZTUCZNEJ INTELIGENCJI (AI)

Dotyczy grupowej pracy inżynierskiej pt. „Wellhicle – platforma do zarządzania wypożyczalniami pojazdów w modelu SaaS", przygotowanej przez studentów kierunku Informatyka Polsko-Japońskiej Akademii Technik Komputerowych.

Każdy z niżej podpisanych współautorów oświadcza, że:

1. zapoznał się z treścią Zarządzenia Rektora Polsko-Japońskiej Akademii Technik Komputerowych z dnia 1 kwietnia 2026 r. oraz z „Wytycznymi dotyczącymi wykorzystywania sztucznej inteligencji w procesie dydaktycznym PJATK", stanowiącymi załącznik do ww. zarządzenia;
2. ewentualne wykorzystanie narzędzi sztucznej inteligencji (AI) w niniejszej pracy miało charakter wyłącznie pomocniczy i nie stanowiło zastępstwa dla samodzielnej pracy;
3. treść pracy stanowi własne, samodzielne opracowanie autorów, którzy ponoszą pełną odpowiedzialność za jej zawartość, niezależnie od wykorzystania narzędzi AI;
4. nie wykorzystywał narzędzi AI do generowania treści stanowiących zasadniczą część pracy, w szczególności analiz, wniosków, interpretacji ani części podlegających ocenie jako przejaw twórczości własnej;
1. w przypadku korzystania z narzędzi AI sporządził Raport użycia AI zgodnie z obowiązującymi Wytycznymi i dołączył go do pracy.

Niniejsze oświadczenie składa, w odniesieniu do całości pracy oraz swojego indywidualnego wkładu, każdy z poniżej podpisanych współautorów:

Lp.	**Imię i nazwisko**	**Nr albumu**	**Czytelny podpis**
1.	Maciej Alfut	s27322	
2.	Kacper Gajdamowicz	s27372	
3.	Michał Gołaszewski	s26525	
4.	Łukasz Lendzion	s27952	
5.	Wojciech Matusiewicz	s27661	
6.	Piotr Minta	s29460	
7.	Maciej Sobolewski	s26519	
8.	Michał Stasik	s27525	

Streszczenie

Praca obejmuje analizę, projekt, implementację i testy Wellhicle – wielotenantowej platformy SaaS do zarządzania wypożyczalniami pojazdów. Istotą rozwiązania jest obsługa wielu niezależnych wypożyczalni na wspólnej platformie, z izolacją ich danych, oraz rozróżnienie klienta platformy, czyli wypożyczalni, od klienta wypożyczalni rezerwującego pojazd. Na system składa się trzynaście modułów pokrywających pełny cykl działalności wypożyczalni, wraz z jej wielojęzyczną, wielowalutową i personalizowaną stroną publiczną. Rozwiązanie tworzą serwer aplikacyjny .NET w czystej architekturze według wzorców DDD i CQRS, aplikacje przeglądarkowe w Angularze ze wspólną biblioteką, wieloplatformowa aplikacja mobilna, własna infrastruktura uruchomieniowa oraz integracja z systemem ERP enova365. Powstały system zweryfikowano testami jednostkowymi, integracyjnymi, automatycznymi i manualnymi. Pracę zamyka omówienie napotkanych problemów i kierunków dalszego rozwoju.

Słowa kluczowe

platforma SaaS; wielotenantowość; wypożyczalnia pojazdów; aplikacja mobilna;

Spis treści

Spis treści 4
1. Wstęp 13
1.1. Cel pracy 13
1.2. Zakres i rezultat prac 13
2. Istniejące rozwiązania 15
2.1. RentSyst 15
2.2. HQ Rental Software 16
2.3. RentHelp.io 18
2.4. FleetWire.io 19
2.5. Porównanie istniejących rozwiązań 20
3. Planowanie i organizacja pracy 21
3.1. Założenia i ograniczenia projektu 22
3.2. Plan pierwotny i pierwsze decyzje organizacyjne 22
3.3. Podział odpowiedzialności 23
3.3.1. Kierownictwo projektu 24
3.3.2. Analiza 24
3.3.3. Projektowanie systemu i architektury 24
3.3.4. Design GUI 25
3.3.5. Backend 25
3.3.6. Frontend web 25
3.3.7. Aplikacja mobilna 26
3.3.8. DevOps i infrastruktura 26
3.3.9. Testowanie 26
3.3.10. Integracja z systemem ERP 26
3.3.11. Dokumentacja końcowa 26
3.4. Narzędzia pracy zespołowej 27
3.4.1. Jira 27
3.4.2. Confluence 30
3.4.3. GitHub 31
3.4.4. Microsoft Teams 32

3.4.5. Discord 32
3.4.6. OneDrive 32
3.4.7. Bot powiadomień z Jiry 33
3.5. Komunikacja i spotkania 33
3.6. Zmiany zakresu projektu 34
3.6.1. Moduł finansowy oraz integracja z systemem ERP 34
3.6.2. Pozostałe zmiany zakresu 35
3.7. Organizacja dokumentacji końcowej 35
3.7.1. Nowy projekt w Jira 35
3.7.2. Obieg dokumentu 37
3.8. Korekty planu i wnioski organizacyjne 37
3.8.1. Korekty planu i przebiegu prac 37
3.8.2. Wnioski organizacyjne 38
4. Narzędzia i technologie 40
4.1. Analiza 40
4.1.1. Lucidchart 40
4.1.2. Draw.io 40
4.1.3. Microsoft Visio 40
4.1.4. Enterprise Architect 41
4.2. Projekt interfejsu użytkownika 42
4.2.1. Figma 42
4.2.2. Inkscape 42
4.2.3. Material Design 3 42
4.2.4. Material Theme Builder 42
4.3. Serwer aplikacyjny 43
4.3.1. Visual Studio i JetBrains Rider 43
4.3.2. Git 43
4.3.3. SQL Server Management Studio 43
4.3.4. Docker i Docker Compose 43
4.3.5. C# 43
4.3.6. .NET 8 44
4.3.7. Microsoft SQL Server 44

4.3.8. Entity Framework Core 44
4.3.9. ASP.NET Core 44
4.3.10. FluentValidation 44
4.3.11. OpenAPI 45
4.3.12. AWSSDK.S3 45
4.3.13. SixLabors.ImageSharp 45
4.3.14. DocumentFormat.OpenXml 45
4.3.15. iText 45
4.3.16. RabbitMQ 46
4.3.17. Hangfire 46
4.3.18. IMemoryCache 46
4.3.19. JWT 46
4.3.20. Microsoft.AspNetCore.Identity PasswordHasher 46
4.3.21. Serilog 47
4.3.22. Microsoft.Extensions.Hosting i Dependency Injection 47
4.3.23. Microsoft.Extensions.Configuration i wzorzec Options 47
4.3.24. Health Checks 47
4.3.25. Polly 47
4.3.26. libphonenumber-csharp 48
4.4. Interfejs przeglądarkowy 48
4.4.1. Angular 20 48
4.4.2. TypeScript 48
4.4.3. RxJS 48
4.4.4. Angular Material 48
4.4.5. ESLint 49
4.4.6. Prettier 49
4.4.7. Ngx-translate 49
4.5. Interfejs mobilny 49
4.5.1. Kotlin Multiplatform 49
4.5.2. Compose Multiplatform 50
4.5.3. Ktor 50
4.5.4. Koin 50

4.5.5. Przechowywanie danych lokalnych - multiplatform-settings 50
4.5.6. BuildKonfig 51
4.5.7. Firebase App Distribution 51
4.5.8. Navigation Compose 51
4.6. Wdrażanie i monitorowanie 51
4.6.1. OpenTelemetry 51
4.6.2. NGINX 52
4.6.3. Kubernetes 52
4.6.4. Cloudflare Tunnel 52
4.6.5. Harbor 52
4.6.6. RustFS 53
4.6.7. Prometheus 53
4.6.8. Loki 53
4.6.9. Tempo 53
4.6.10. Grafana 54
4.6.11. K3s 54
4.6.12. Helm 54
4.6.13. GitHub Actions 54
4.6.14. Kustomize 55
4.7. Testy 55
4.7.1. Postman 55
4.7.2. Insomnia 55
4.7.3. Swagger 56
4.7.4. Scalar 56
4.7.5. Playwright 58
4.7.6. Biblioteka xUnit 58
5. Analiza 60
5.1. Streszczenie 61
5.2. Aktorzy systemu 61
5.2.1. Tabela zbiorcza aktorów 62
5.2.2. Aktorzy globalni 63
5.2.3. Personel wypożyczalni 64

5.2.4. Klienci wypożyczalni 64
5.2.5. Aktorzy techniczni 65
5.2.6. Dziedziczenie aktorów 65
5.3. Zakres funkcjonalny 66
5.3.1. Zarządzanie wypożyczalnią 67
5.3.2. Flota 70
5.3.3. Klienci 72
5.3.4. Oferty 74
5.3.5. Rezerwacje 77
5.3.6. Finanse 81
5.3.7. Płatności 86
5.3.8. Operacyjny 90
5.3.9. Strona publiczna wypożyczalni 92
5.3.10. Pakiety 94
5.3.11. Administracja platformy 97
5.3.12. Komunikacja 100
5.3.13. Strona publiczna platformy 102
5.4. Wymagania niefunkcjonalne 103
5.4.1. Bezpieczeństwo i ochrona danych 103
5.4.2. Wydajność i skalowalność 104
5.4.3. Niezawodność 104
5.4.4. Użyteczność interfejsów 105
5.4.5. Wielojęzyczność i lokalizacja 105
5.4.6. Utrzymanie i monitorowanie systemu 105
5.4.7. Kompatybilność i przenośność 106
5.5. Przypadki użycia 107
5.5.1. Zarządzanie wypożyczalnią 107
5.5.2. Flota 108
5.5.3. Klienci 108
5.5.4. Oferty 109
5.5.5. Rezerwacje 110
5.5.6. Płatności 112

5.5.7. Operacyjny 113
5.5.8. Strona publiczna wypożyczalni 114
5.5.9. Pakiety 115
5.5.10. Administracja platformy 116
5.5.11. Komunikacja 116
5.5.12. Strona publiczna platformy 117
5.6. Wybrane scenariusze i diagramy aktywności 118
5.6.1. Generyczne wzorce aktywności 118
5.6.2. Anonimizacja danych klienta wypożyczalni 130
5.6.3. Przeglądanie historii rezerwacji przez klienta 132
5.6.4. Tworzenie rezerwacji w imieniu klienta 134
5.6.5. Wydanie pojazdu 136
5.6.6. Odbiór pojazdu 139
5.6.7. Zgłoszenie usterki 142
5.6.8. Zmiana własnej domeny 143
5.7. Analityczny diagram klas 145
5.7.1. Platforma SaaS 148
5.7.2. Flota 149
5.7.3. Oferty i rezerwacje 150
5.7.4. Finanse i płatności 151
5.8. Podsumowanie 152
6. Projektowanie 153
6.1. Założenia projektowe 153
6.1.1. Wielotenantowość 153
6.1.2. Wielojęzyczność 153
6.1.3. Wielowalutowość 154
6.1.4. Podział aplikacji *frontendowych* na 4 oddzielne serwery WWW 154
6.1.5. Integracja z systemem ERP 155
6.1.6. Wieloplatformowa aplikacja mobilna 155
6.2. Architektura systemu 155
6.2.1. Serwer *proxy* 156
6.2.2. Serwer WWW 156

6.2.3. Serwer aplikacyjny 157
6.2.4. Baza danych 157
6.2.5. Uwierzytelnianie i autoryzacja 158
6.2.6. Serwer plików 159
6.2.7. Broker wiadomości 159
6.2.8. Serwer wysyłania wiadomości e-mail 159
6.2.9. Generator dokumentów 160
6.2.10. Aplikacja mobilna 161
6.3. Projektowy diagram klas 161
6.3.1. Wspólne podstawy domeny 162
6.3.2. Obiekty wartości 162
6.3.3. Diagramy modułowe 163
6.3.4. Enumeracje 176
6.4. Projekt interfejsu użytkownika 182
6.4.1. Nazwa i logo systemu 182
6.4.2. Założenia projektowe interfejsu 184
6.4.3. Projekt wybranych widoków systemu 184
6.4.4. Zakres i rola przygotowanego projektu interfejsu 196
7. Implementacja 197
7.1. Serwer aplikacyjny 197
7.1.1. Struktura rozwiązania w .NET 197
7.1.2. Warstwa domeny 199
7.1.3. Warstwa aplikacji – wzorzec CQRS 206
7.1.4. Warstwa infrastruktury – dostęp do danych 211
7.1.5. Zabezpieczenia i autoryzacja 216
7.1.6. Obsługa błędów 218
7.1.7. Obserwowalność 220
7.1.8. Procesy w tle 221
7.1.9. Przechowywanie plików biznesowych 224
7.1.10. Generowanie i przetwarzanie dokumentów 231
7.1.11. Integracje 238
7.2. Interfejs przeglądarkowy 242

7.2.1. Bezpieczeństwo ... 242
7.2.2. Biblioteka elementów powszechnie używanych w aplikacjach ... 244
7.2.3. Aplikacja do prezentacji oferty systemu ... 247
7.2.4. Aplikacja do zarządzania wypożyczalnią ... 249
7.2.5. Aplikacja do administrowania wypożyczalniami ... 278
7.2.6. Aplikacja do korzystania z usług wypożyczalni ... 280
7.3. Interfejs mobilny ... 289
7.3.1. Nawigacja ... 289
7.3.2. Struktura aplikacji ... 291
7.3.3. Konwencjonalne wtyczki Gradle ... 292
7.3.4. Procesy biznesowe ... 293
7.3.5. Ciekawe rozwiązania ... 300
7.4. Wdrażanie i monitorowanie ... 305
7.4.1. Konfiguracja środowiska uruchomieniowego ... 305
7.4.2. Proces integracji kodu ... 309
7.4.3. Proces wdrażania kodu ... 310
7.4.4. Ciekawe rozwiązania ... 312
7.5. Integracja z systemem ERP enova365 ... 312
7.5.1. System enova365 i model rozszerzeń ... 313
7.5.2. Założenia i zakres integracji ... 313
7.5.3. Architektura dodatku ... 313
7.5.4. Model danych i mapowanie na encje enova365 ... 315
7.5.5. Procesy synchronizacji i wymiana dokumentów ... 316
7.5.6. Konfiguracja, uruchomienie i automatyzacja ... 317
7.5.7. Stan rozwiązania i kierunki rozwoju ... 317
8. Testy ... 319
8.1. Testy manualne i integracyjne ... 319
8.1.1. Testy integracyjne ... 320
8.1.2. Testy manualne ... 321
8.2. Testy automatyczne ... 323
8.3. Testy jednostkowe ... 328
9. Problemy związane z systemem ... 332

10. Plany na przyszłość 334
11. Podsumowanie 337
12. Dodatki 338
12.1. Wykorzystanie narzędzi sztucznej inteligencji 338
12.2. Słownik pojęć 343
12.3. Spis tabel 344
12.4. Spis rysunków 347
12.5. Spis listingów 351
12.6. Cytowane prace 355

1. Wstęp

Wellhicle to platforma udostępniana w modelu SaaS (ang. *Software as a Service*), służąca do kompleksowego prowadzenia wypożyczalni pojazdów. Praca dokumentuje jej powstanie – od analizy potrzeb i istniejących rozwiązań, przez projekt i implementację, po testy oraz wdrożenie gotowego systemu.

1.1. Cel pracy

Celem pracy było zaprojektowanie i wspólna realizacja dużego, wielowarstwowego systemu informatycznego – platformy działającej w modelu SaaS – o skali i złożoności zbliżonej do rozwiązań komercyjnych. Dziedziną, w której platformę zrealizowano, jest zarządzanie wypożyczalniami pojazdów. Wybór tej dziedziny wynikał z analizy dostępnych na rynku systemów tej klasy: większość z nich ogranicza się do panelu administracyjnego oraz widżetów osadzanych na stronie wypożyczalni i nie zapewnia ani pełnowartościowej, możliwej do personalizacji witryny dla klienta wypożyczalni, ani obsługi wielu niezależnych wypożyczalni w obrębie jednej platformy.

W wymiarze biznesowym celem było dostarczenie kompletnej platformy, która pozwala wielu niezależnym wypożyczalniom prowadzić działalność bez budowania własnego oprogramowania. Platforma pokrywa zarówno zaplecze operacyjne wypożyczalni (m.in.: flotę, oferty, rezerwacje, rozliczenia oraz procesy wydania i zwrotu pojazdu) jak i publiczną stronę, na której klienci wypożyczalni przeglądają ofertę i składają rezerwacje. Dla całego rozwiązania kluczowe jest oddzielenie dwóch perspektyw: klienta platformy, czyli wypożyczalni wykupującej dostęp do systemu oraz klienta wypożyczalni, czyli osoby rezerwującej pojazd.

W wymiarze inżynierskim celem było zaprojektowanie i zbudowanie systemu o architekturze odpowiadającej rzeczywistym potrzebom takiej platformy: wielomodułowego serwera aplikacyjnego, odrębnych aplikacji przeglądarkowych dla różnych grup odbiorców, aplikacji mobilnej dla procesów terenowych oraz infrastruktury zapewniającej wdrażanie, monitorowanie i utrzymanie całości.

1.2. Zakres i rezultat prac

Zakres projektu wykracza poza pojedynczą aplikację. System tworzy kilka współpracujących komponentów dostosowanych do różnych grup odbiorców: panel do zarządzania wypożyczalniami, interfejs do korzystania z usług wypożyczalni, portal do administrowania wypożyczalniami oraz stronę prezentującą oferty systemu. Uzupełnia je aplikacja mobilna. Funkcjonalność systemu obejmuje trzynaście modułów: od zarządzania flotą, ofertami i rezerwacjami, przez rozliczenia i płatności, dokumenty oraz komunikację z klientami, po subskrypcje, pakiety i administrację całą platformą. Komponenty klienckie są dostępne w wielu wersjach językowych i obsługują wiele walut.

Rezultatem prac jest działający i wdrożony system. Warstwa serwerowa została zbudowana w technologii .NET, zgodnie z zasadami architektury wielowarstwowej; realizuje on logikę biznesową i udostępnia ją aplikacjom klienckim poprzez interfejs programistyczny. Serwisy przeglądarkowe powstały w Angularze, wykorzystując wspólną bibliotekę ograniczającą powielanie kodu. Aplikacja mobilna została zaimplementowana jako rozwiązanie wieloplatformowe, oparte na jednym wspólnym kodzie dla systemów Android i iOS. Całość działa na własnej infrastrukturze opartej na konteneryzacji i orkiestracji, z procesami ciągłej integracji i wdrażania oraz rozproszonym monitorowaniem. Poprawność rozwiązania potwierdzono testami manualnymi, integracyjnymi, automatycznymi oraz jednostkowymi. System uzupełniono integracją z systemem ERP enova365, odpowiadającą za formalne rozliczenie rezerwacji i subskrypcji.

Skala przedsięwzięcia odpowiada jego założeniom: prace prowadził ośmioosobowy zespół przez około osiem miesięcy, a ich efektem jest spójny, wielomodułowy system pokrywający pełny cykl działalności wypożyczalni – od konfiguracji organizacji, przez obsługę klienta wypożyczalni, po rozliczenia i procesy terenowe.

2. Istniejące rozwiązania

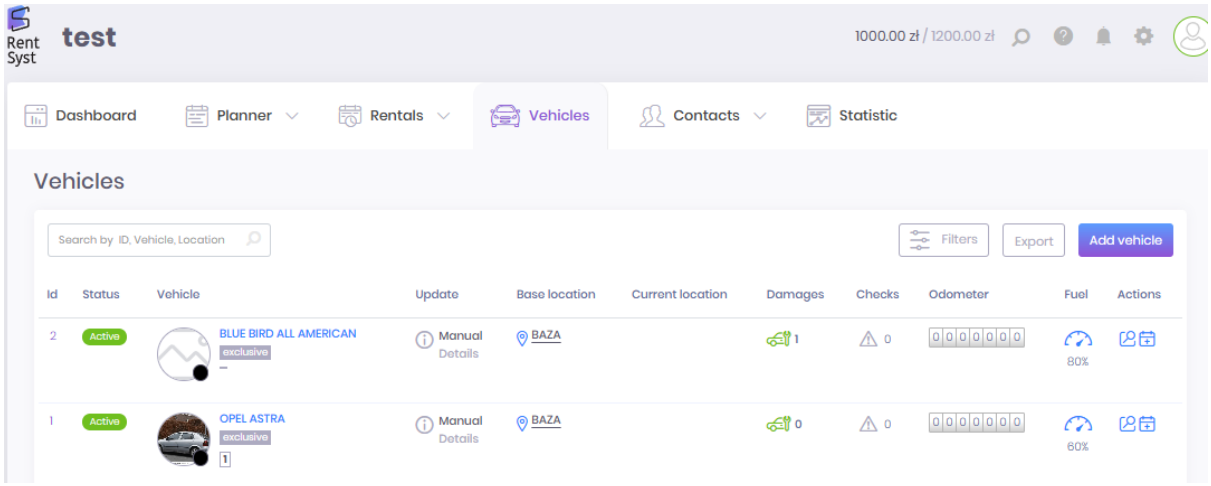
W rozdziale tym przedstawiono i porównano wybrane rozwiązania informatyczne wspierające działalność wypożyczalni pojazdów, określając jednocześnie zakres oferowanych przez nie funkcji. Analiza stanowiła punkt odniesienia dla dalszych prac nad przygotowaniem założeń systemu Wellhicle, w szczególności w zakresie obsługi floty, rezerwacji, klientów, płatności, dokumentów oraz procesów operacyjnych.

Kryteria porównania dobrano tak, aby obejmowały najważniejsze obszary działania systemu dla wypożyczalni pojazdów: obsługę operacyjną, zarządzanie procesami oraz sposób udostępniania oferty klientowi wypożyczalni. Wynikają one z logiki pracy wypożyczalni oraz z założeń projektu Wellhicle, który ma wspierać zarówno wewnętrzną administrację, jak i publiczną prezentację usług.

Zakres porównania obejmuje cztery rozwiązania: RentSyst, HQ Rental Software, Renthelp.io oraz FleetWire.io. Porównanie przedstawia stan ww. systemów z października 2025.

2.1. RentSyst

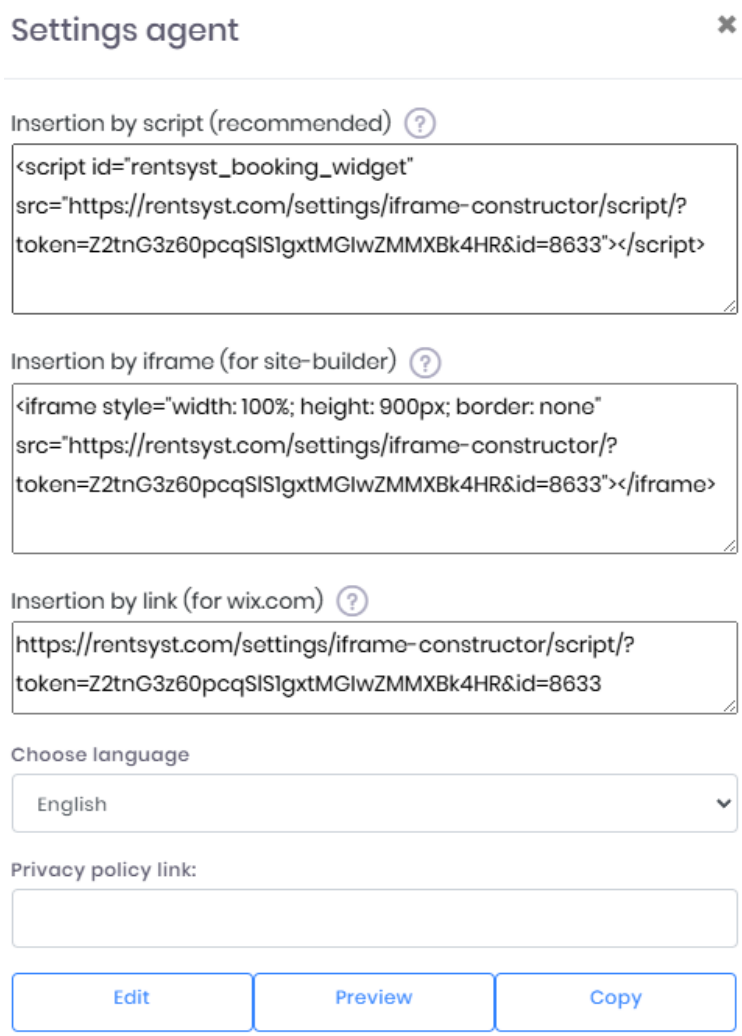
RentSyst [1] jest chmurowym systemem przeznaczonym dla przedsiębiorstw świadczących usługi wynajmu pojazdów. System umożliwia prowadzenie ewidencji pojazdów, rejestrowanie usterek, obsługę rezerwacji, zarządzanie kontami klientów i pracowników oraz obsługę wielu oddziałów. Przykładowy widok panelu operacyjnego, obejmujący pracę z flotą, przedstawiono na *Rysunek 1*.



Rysunek 1 Panel operacyjny RentSyst z widokiem floty. Źródło: opracowanie własne.

RentSyst udostępnia moduł raportów i statystyk, które obejmują statusy wynajmów, przychody i koszty wypożyczalni. System obsługuje także integrację z dostawcami płatności online, powiadomienia oraz korzystanie z szablonów dokumentów, przy czym tworzenie dokumentów opiera się na już dostępnych szablonach.

W zakresie kontaktu z klientem wypożyczalni RentSyst nie dostarcza gotowej publicznej strony internetowej wypożyczalni. Umożliwia natomiast integrację z istniejącą stroną internetową, która realizowana jest poprzez widżet gotowy do umieszczenia na stronie internetowej wypożyczalni. Przykład konfiguracji wykorzystywanej do połączenia systemu z zewnętrzną stroną przedstawiono na *Rysunek 2*.

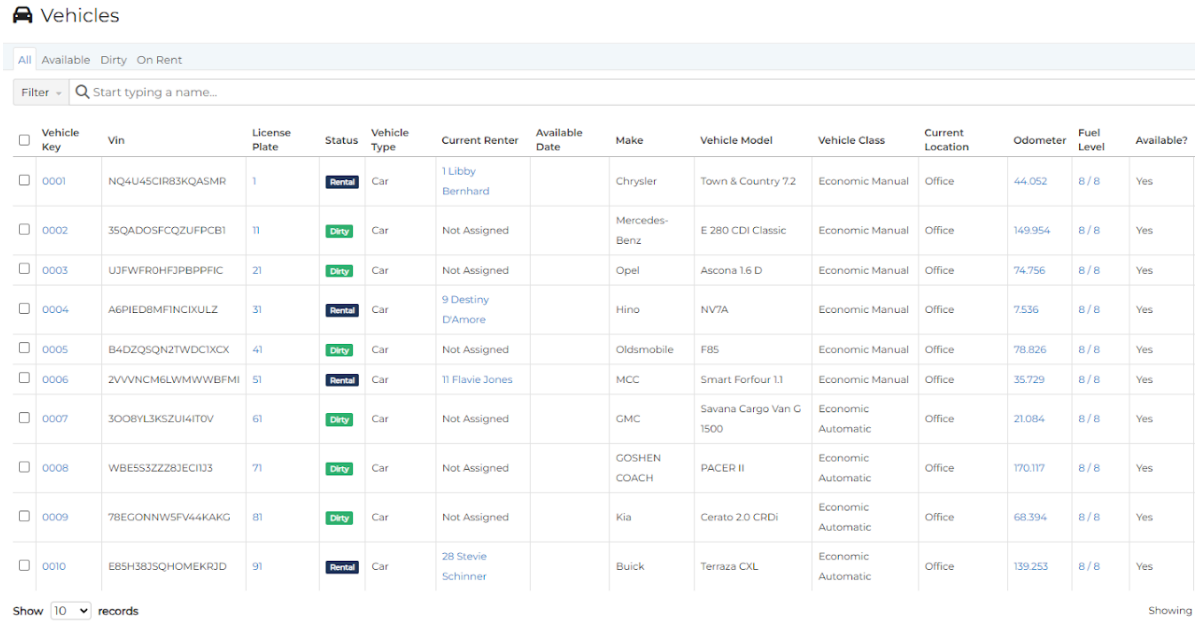


Rysunek 2 Widok konfiguracji widżetów RentSyst do integracji z własną stroną. Źródło: opracowanie własne.

2.2. HQ Rental Software

HQ Rental Software [2] jest chmurowym systemem klasy CRM przeznaczonym dla firm zajmujących się wynajmem pojazdów i innego sprzętu. Jest to rozwiązanie wspierające codzienną pracę wypożyczalni przez integrację obsługi floty, klientów, pracowników, rezerwacji, dokumentów i płatności.

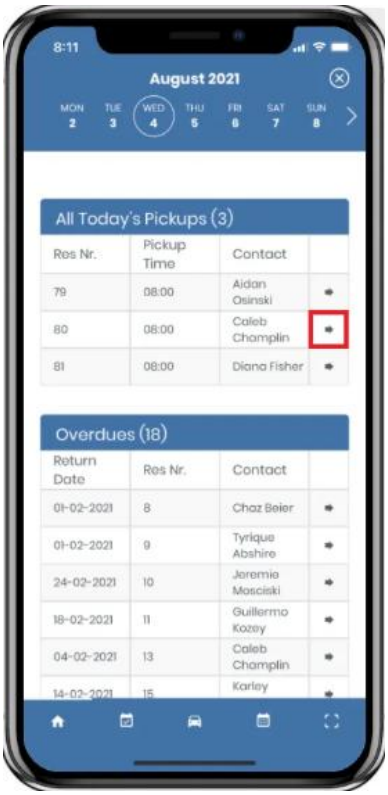
System umożliwia zarządzanie flotą pojazdów, kontami klientów i pracowników, rezerwacjami, usterkami oraz wieloma oddziałami. Przykładowy panel zarządzania flotą przedstawiono na *Rysunek 3*.



Rysunek 3 Fragment panelu operacyjnego HQ Rental Software z widokiem floty. Źródło: opracowanie własne.

Istotnym elementem HQ Rental Software jest rozbudowany moduł raportów i statystyk, który wyróżnia system na tle części pozostałych analizowanych systemów. Rozwiązanie wspiera także tworzenie szablonów dokumentów, integrację z systemami płatności oraz system powiadomień.

HQ Rental Software jest jedynym systemem z analizowanych, który posiada dedykowaną aplikację mobilną. Przykładowy widok aplikacji mobilnej, istotny z perspektywy pracy operacyjnej poza stanowiskiem komputerowym, przedstawiono na *Rysunek 4*.

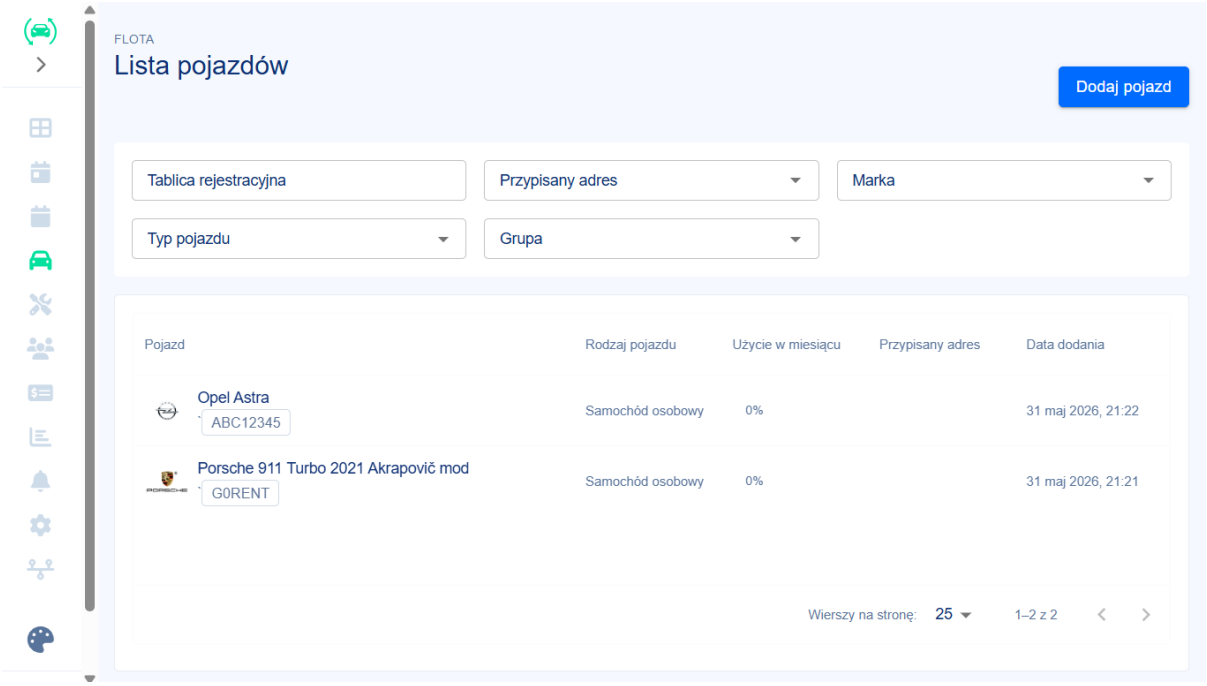


Rysunek 4 Widok aktywnych odbiorów w aplikacji mobilnej HQ Rental Software. Źródło: opracowanie własne.

W zakresie kontaktu z klientem wypożyczalni, HQ Rental Software nie dostarcza samodzielnej, gotowej strony publicznej wypożyczalni jako integralnego elementu systemu. Możliwa jest współpraca z zewnętrznym kreatorem stron Caribonix (jest to zewnętrzne narzędzie typu CMS, niepowiązane bezpośrednio z HQ Rental Software), a także integracja systemu z istniejącą stroną internetową poprzez widżet lub wtyczkę do systemu WordPress CMS.

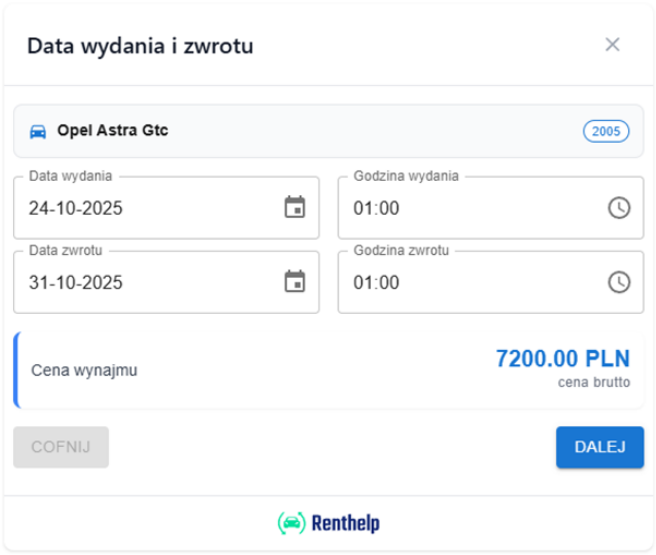
2.3. RentHelp.io

RentHelp.io [3] jest aplikacją SaaS kierowaną do firm prowadzących wypożyczalnie samochodów oraz innych pojazdów, takich jak auta dostawcze, kampery, motocykle czy przyczepy. System koncentruje się na obsłudze podstawowych procesów operacyjnych wypożyczalni, w szczególności zarządzaniu flotą, rezerwacjami, klientami i dokumentacją działalności. Widok panelu zarządzania flotą systemu przedstawiono na *Rysunek 5*.



Rysunek 5 Panel operacyjny RentHelp.io z widokiem floty. Źródło: opracowanie własne.

RentHelp.io nie udostępnia gotowej strony publicznej wypożyczalni, ani podłączenia własnej domeny. Rozwiązanie nie posiada dedykowanej aplikacji mobilnej. Wspiera natomiast integrację z systemami płatności oraz integrację z istniejącą stroną internetową wypożyczalni poprzez widżet do osadzenia na własnej stronie internetowej. Kontakt z klientem wypożyczalni może być więc realizowany przez stronę właściciela wypożyczalni z wykorzystaniem elementów integracyjnych systemu. Przykład takiej integracji przedstawiono na *Rysunek 6*.

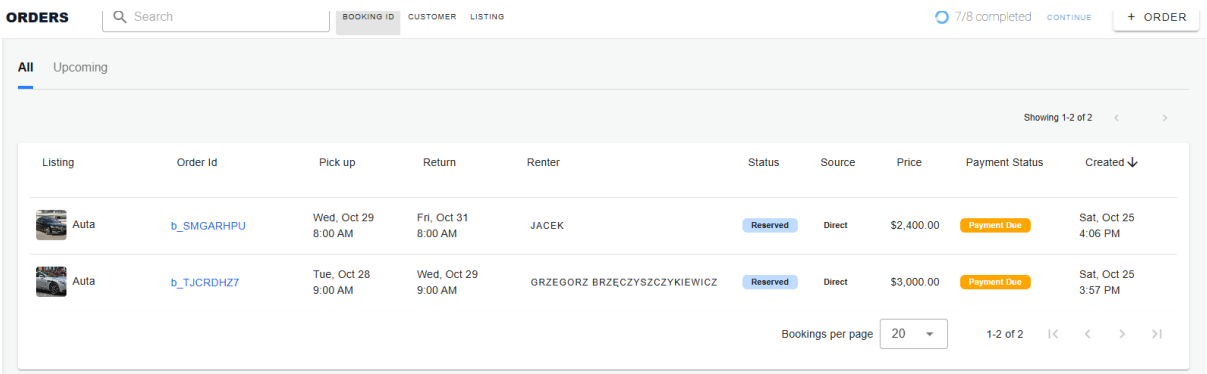


Rysunek 6 Widżet rezerwacji w RentHelp.io, do osadzenia na własnej stronie internetowej. Źródło: opracowanie własne.

2.4. FleetWire.io

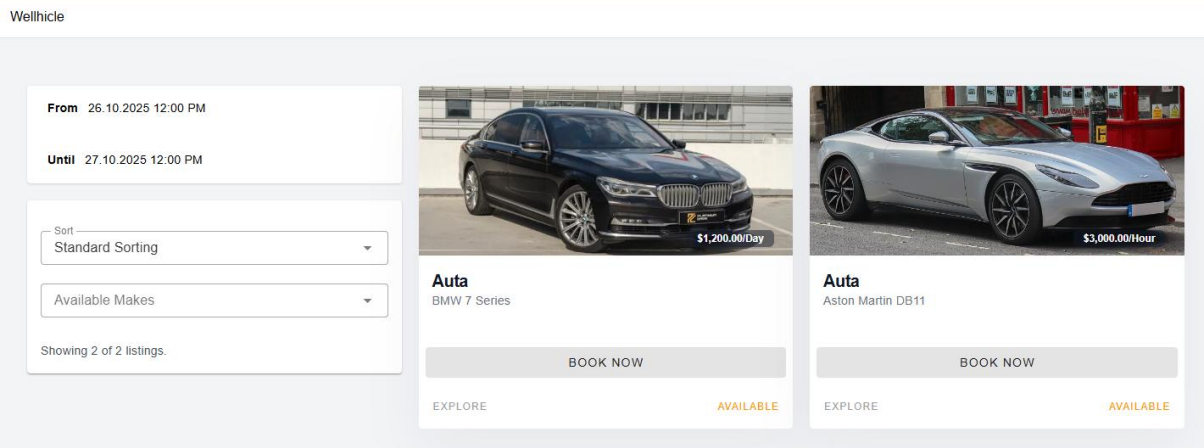
FleetWire.io [4] jest chmurowym systemem klasy CRM dla wypożyczalni pojazdów. Rozwiązanie jest kierowane przede wszystkim do firm, które potrzebują narzędzia do obsługi floty, klientów, pracowników, rezerwacji, dokumentów oraz płatności w jednym środowisku.

System umożliwia zarządzanie flotą pojazdów, kontami klientów i pracowników, rezerwacjami, usterkami oraz wieloma oddziałami. Wspiera także integrację z systemami płatności, dodawanie i tworzenie szablonów dokumentów oraz system powiadomień. Przykładowy widok panelu operacyjnego FleetWire.io przedstawiono na *Rysunek 7*.



Rysunek 7 Panel operacyjny FleetWire.io z widokiem rezerwacji. Źródło: opracowanie własne.

W odróżnieniu od RentSyst, HQ Rental Software i RentHelp.io, FleetWire.io udostępnia stronę internetową wypożyczalni, natomiast bez możliwości jej personalizacji. System umożliwia również podłączenie własnej domeny oraz integrację z istniejącą stroną internetową poprzez widżet, wtyczkę do systemu WordPress CMS oraz wystawia API. Przykładową stronę publiczną wypożyczalni udostępnianą w ramach FleetWire.io przedstawiono na *Rysunek 8*.



Rysunek 8 Widok domyślnej strony internetowej FleetWire.io. Źródło: opracowanie własne.

2.5. Porównanie istniejących rozwiązań

Zestawienie syntetyzuje wyniki wcześniejszej analizy wybranych systemów wspierających działalność wypożyczalni pojazdów (Tabela 1). Porównanie obejmuje najważniejsze funkcje istotne z perspektywy organizacji pracy wypożyczalni oraz założeń projektowych systemu Wellhicle.

Tabela 1 Zestawienie istniejących rozwiązań. Źródło: opracowanie własne.

Funkcjonalność	**Wellhicle - plan**	**RentSyst**	**HQ Rental Software**	**FleetWire.io**	**RentHelp.io**
Zarządzanie flotą pojazdów	Tak	Tak	Tak	Tak	Tak
Zarządzanie kontami klientów	Tak	Tak	Tak	Tak	Tak
Zarządzanie kontami pracowników	Tak	Tak	Tak	Tak	Tak
Moduł raportów i statystyk	Tak	Tak - mało zaawansowany	Tak - zaawansowany	Nie	Nie
Zarządzanie i edycja rezerwacji pojazdów	Tak	Tak	Tak	Tak	Tak
Dedykowana aplikacja mobilna	Tak	Nie	Tak	Nie	Nie
Strona internetowa wypożyczalni	Tak	Nie	Nie	Tak	Nie

Personalizacja dedykowanej strony internetowej	Tak	Nie	Nie	Nie	Nie
Podłączenie własnej domeny	Tak	Nie	Tak	Tak	Nie
Integracja systemu z własną stroną internetową	Nie	Tak	Tak	Tak	Tak
Obsługa płatności	Tak	Tak	Tak	Tak	Tak
Dodawanie i tworzenie szablonów dokumentów	Tak	Tak - tylko z już wgranych szablonów	Tak	Tak	Nie
Rejestracja usterek pojazdów	Tak	Tak	Tak	Tak	Tak
Dodanie i obsługa wielu oddziałów	Tak	Tak	Tak	Tak	Nie
System powiadomień	Tak	Tak	Tak	Tak	Nie
Darmowa wersja próbna	Tak	Tak	Tak	Tak	Tak

Wyniki porównania wskazują, że analizowane systemy w znacznym stopniu pokrywają podstawowe funkcje operacyjne wypożyczalni, jednak różnią się zakresem rozwiązań dotyczących strony internetowej wypożyczalni, personalizacji, aplikacji mobilnej, raportów oraz sposobu obsługi klienta wypożyczalni. W badanej próbie żaden z porównywanych systemów nie łączy wszystkich analizowanych cech, dlatego zestawienie stanowi punkt odniesienia dla określenia zakresu funkcjonalnego projektowanego systemu.

3. Planowanie i organizacja pracy

Planowanie w projekcie Wellhicle było niezbędnym elementem organizacyjnym, wynikającym ze skali przedsięwzięcia, liczby uczestników prac, ograniczonego czasu realizacji oraz formalnych wymogów związanych z przygotowaniem pracy inżynierskiej. Zakres projektu obejmował typowe dla dużych systemów IT etapy: analizę wymagań, architekturę, implementację warstwy serwerowej i aplikacji interfejsu użytkownika (zarówno przeglądarkowych, jak i mobilnych), infrastrukturę uruchomieniową oraz kompleksowe testy. W tych okolicznościach planowanie pełniło kluczową rolę

jako proces ciągły: porządkowało zadania, pozwalało na efektywny podział obowiązków i ułatwiało komunikację dotyczącą postępów prac.

Punktem wyjścia dla planowania były pierwsze spotkania zespołu w październiku 2025 r., a jego wstępny kształt został udokumentowany w szczegółowym konspekcie pracy. W trakcie realizacji planowanie nie ograniczało się do jednego dokumentu - kolejne ustalenia zapadały podczas regularnych spotkań oraz były wynikiem decyzji kierownika projektu i liderów poszczególnych obszarów. Rzeczywistą wizualizację postępów zapewniał system Jira, cykliczne spotkania statusowe oraz dokumentacja współdzielona za pośrednictwem platform Confluence i OneDrive.

3.1. Założenia i ograniczenia projektu

Projekt został zrealizowany przez ośmioosobowy zespół w ramach pracy inżynierskiej, w związku z czym głównymi ograniczeniami były czas przeznaczony na przygotowanie rozwiązania oraz brak budżetu projektowego. Jednocześnie zakres systemu był szeroki: obejmował platformę typu oprogramowanie jako usługa - SaaS (ang. *Software as a Service*) do obsługi wypożyczalni pojazdów, panel operacyjny wypożyczalni, stronę publiczną wypożyczalni, panel administracji platformy, aplikację mobilną oraz zaplecze infrastrukturalne. Taki zakres wymagał podziału prac na obszary oraz mechanizmów pozwalających zespołowi śledzić aktualny stan prac w każdym z nich.

Istotnym ograniczeniem była niepewność typowa dla początkowej fazy projektu. Na etapie pierwszych spotkań zespół nie posiadał jeszcze pełnej perspektywy dotyczącej przyszłego zakresu systemu, szczegółowych wymagań funkcjonalnych ani zależności między modułami. Planowanie musiało więc jednocześnie porządkować bieżące ustalenia i pozostawiać przestrzeń na późniejsze korekty.

Na sposób planowania wpływały również wymagania promotora oraz organizacja zajęć dyplomowych. Przygotowanie konspektu pracy, przedstawianie postępów i określanie planów na kolejne tygodnie wymuszały regularną kontrolę kierunku prac. Podczas spotkań realizowanych w trakcie zjazdów na uczelni przedstawiano stan projektu i plan na następny okres, a krótkie ustalenia były zapisywane w notatkach roboczych. Był to ważny mechanizm kontroli, choć nie zastępował pełnego, aktualizowanego harmonogramu.

Prace były prowadzone równolegle z innymi obowiązkami uczestników projektu. Ten kontekst miał znaczenie organizacyjne i tłumaczy wykorzystanie zarówno formalnych narzędzi projektowych, jak i szybkich kanałów komunikacji.

3.2. Plan pierwotny i pierwsze decyzje organizacyjne

Wczesny etap planowania, prowadzony w październiku 2025 r., rozpoczął się od ustalenia zakresu i kierunku projektu. Punktem wyjścia była analiza dostępnych na rynku systemów do zarządzania wypożyczalniami pojazdów. Z analizy wynikło, że większość rozwiązań ogranicza się jedynie do panelu administracyjnego oraz widżetów przeznaczonych do osadzania na stronie klienta, nie zapewniając pełnowartościowej strony dla klienta wypożyczalni. Ta obserwacja wyznaczyła kierunek projektu – Wellhicle miał być kompletną platformą SaaS, obejmującą zarówno zaplecze operacyjne wypożyczalni, jak i stronę klienta wypożyczalni.

Kolejną decyzją było wyodrębnienie podstawowych komponentów systemu: panel dla personelu wypożyczalni, stronę klienta z obsługą rezerwacji, panel administracyjny platformy oraz aplikację mobilną do procesów operacyjnych. Priorytet nadano funkcjom zarządzania rezerwacjami, flotą pojazdów, klientami i integracji systemu płatności oraz generowaniu dokumentów. Równolegle podjęto kluczowe decyzje narzędziowe: do zarządzania zadaniami wybrano Jirę, dokumentację roboczą

w Confluence, przechowywanie kodu na GitHubie, codzienną komunikację przez Discorda oraz oficjalne spotkania za pośrednictwem Microsoft Teams.

W ramach tego samego etapu ustalono zakres minimalnej wersji produktu – MVP (ang. *Minimal Viable Product*) oraz role użytkowników. Ich szczegółowy opis znajduje się w rozdziale *5*. Sformalizowano podział pracy na zespoły, omówiony szerzej w podrozdziale *3.3*. Wprowadzono stałe spotkania statusowe (opisane w podrozdziale *3.5*). Wybrano nazwę projektu „Wellhicle" i zarejestrowano odpowiadające jej domeny: '.pl', '.com' oraz '.com.pl'.

Powstały na tym etapie plan pierwotny utrwalono w konspekcie pracy, który stanowi jego formalny zapis. W zakresie istotnym dla planowania konspekt zawiera trzy główne elementy: harmonogram projektu (fazy: przygotowanie, analiza, projektowanie, implementacja, testy, wdrożenie i dokumentacja), merytoryczny plan pracy z produktami poszczególnych etapów (m.in.: specyfikacja wymagań, architektura systemu, działające API, aplikacje webowa i mobilna, raport testowy oraz tekst pracy) oraz podział odpowiedzialności. Tak ujęty plan pierwotny stał się punktem wyjścia dla dalszych etapów projektu.

3.3. Podział odpowiedzialności

Podział odpowiedzialności w projekcie Wellhicle łączył strukturę sformalizowaną na etapie planowania z faktycznym przebiegiem prac. Początkowy układ wyznaczył obszary i ich liderów, jednak ze względu na skalę projektu, zakresy odpowiedzialności często się nakładały, dlatego zależności pomiędzy poszczególnymi obszarami wymagały bieżących konsultacji oraz punktowego wsparcia między zespołami. *Tabela 2* porządkuje obszary prac, ich formalnych liderów oraz członków zespołów; dalsza część sekcji opisuje każdy obszar osobno, z wyszczególnieniem zakresu obowiązków tam, gdzie zespół liczył co najmniej dwie osoby.

Tabela 2 Podział odpowiedzialności w projekcie Wellhicle. Źródło: opracowanie własne.

Obszar prac	**Formalny lider**	**Członkowie zespołu**
Kierownictwo projektu	Kacper Gajdamowicz	-
Analiza	Kacper Gajdamowicz	Maciej Alfut, Michał Stasik, Michał Gołaszewski, Wojciech Matusiewicz
Projektowanie systemu i architektury	Kacper Gajdamowicz (koordynacja)	Michał Gołaszewski, Maciej Sobolewski, Piotr Minta, Łukasz Lendzion
Design GUI	Maciej Alfut	-
Backend	Maciej Sobolewski	Kacper Gajdamowicz, Wojciech Matusiewicz
Frontend web	Piotr Minta	Michał Gołaszewski, Michał Stasik, Maciej Alfut, Łukasz Lendzion
Aplikacja mobilna	Łukasz Lendzion	-
DevOps i infrastruktura	Michał Gołaszewski	-
Testowanie	Michał Stasik	Łukasz Lendzion, Wojciech Matusiewicz, Maciej Sobolewski, Michał Gołaszewski
Integracja z systemem ERP (enova365)	Kacper Gajdamowicz	-

Dokumentacja końcowa	Wojciech Matusiewicz	wszyscy uczestnicy projektu

3.3.1. Kierownictwo projektu

Kierownictwo projektu prowadził Kacper Gajdamowicz. Do zadań tej roli należała koordynacja zespołów, planowanie i bieżąca priorytetyzacja prac, prowadzenie spotkań statusowych oraz utrzymanie porządku w Jirze. Z perspektywy organizacyjnej kierownictwo łączyło koordynację z istotnym wkładem merytorycznym w wielu obszarach – wnioski wynikające z tego ułożenia opisano w punkcie *3.8.1*.

3.3.2. Analiza

Zespół analityczny prowadził prace nad wymaganiami funkcjonalnymi i niefunkcjonalnymi, aktorami systemu, przypadkami użycia, scenariuszami, diagramami analitycznymi oraz analizą rynku. Wyniki tego obszaru stanowiły punkt odniesienia dla późniejszych decyzji projektowych i implementacyjnych.

Tabela 3 Zakres obowiązków w zespole analitycznym. Źródło: opracowanie własne.

Osoba	**Zakres obowiązków w zespole**
Kacper Gajdamowicz (lider)	zakres systemu, wymagania funkcjonalne, weryfikacja diagramów analitycznych, diagram klas
Maciej Alfut	przypadki użycia, diagramy aktywności, scenariusze, opis aktorów systemu, przykładowe dokumenty umów i regulaminów
Michał Stasik	przypadki użycia, diagramy aktywności, scenariusze, analiza danych obiektów systemu, analiza rynku
Michał Gołaszewski	analiza rynku, analiza integracji płatniczych
Wojciech Matusiewicz	przypadki użycia, diagramy aktywności, scenariusze, wymagania niefunkcjonalne, analiza rynku

3.3.3. Projektowanie systemu i architektury

Obszar obejmował decyzje architektoniczne dotyczące podziału systemu na komponenty, modelu danych, mechanizmów uwierzytelniania oraz zależności między aplikacjami klienckimi a warstwą serwerową. Projekt architektoniczny systemu został zamknięty zgodnie z pierwotnym planem na etapie projektowania. Późniejsze rozszerzenia zakresu były dodawane jako odrębne moduły wpisujące się w istniejącą architekturę.

Tabela 4 Zakres obowiązków w zespole projektowania systemu i architektury. Źródło: opracowanie własne.

Osoba	**Zakres obowiązków w zespole**
Kacper Gajdamowicz (koordynacja)	koordynacja etapu projektowania oraz nadzór nad spójnością decyzji architektonicznych między obszarami
Michał Gołaszewski	architektura rozwiązania, podstawy infrastruktury
Maciej Sobolewski	architektura backendu, warstwy aplikacji, autoryzacja, automatyzacje
Piotr Minta	architektura aplikacji frontendowych i biblioteki wspólnej

Łukasz Lendzion	architektura aplikacji mobilnej

3.3.4. Design GUI

Obowiązki zespołu designu realizował samodzielnie Maciej Alfut. Obszar obejmował makiety, identyfikację wizualną, przepływy użytkownika oraz kierunek wyglądu aplikacji.

3.3.5. Backend

Zespół *backend* odpowiadał za serwer aplikacyjny, API, logikę biznesową, autoryzację, generator dokumentów, magazyn plików, automatyzacje oraz integracje.

Tabela 5 Zakres obowiązków w zespole backendowym. Źródło: opracowanie własne.

Osoba	**Zakres obowiązków w zespole**
Maciej Sobolewski (lider)	struktura backendu, autoryzacja, warstwa aplikacji i domeny, automatyzacje, walidacje, dokumentacja backendu
Kacper Gajdamowicz	implementacja wybranych modułów *backendu* (logika biznesowa, punkty końcowe API), integracja z Cloudflare oraz systemem ERP
Wojciech Matusiewicz	generator dokumentów, magazyn plików, szablony dokumentów, wybrane punkty końcowe API

3.3.6. Frontend web

Zespół *frontend* odpowiadał za aplikacje przeglądarkowe oraz wspólne mechanizmy wykorzystywane między nimi. Formalnie był to jeden zespół, lecz w praktyce prace rozłożyły się między cztery aplikacje *frontendowe* wraz z biblioteką elementów powszechnie używanych w aplikacjach.

Tabela 6 Zakres obowiązków w zespole frontendowym. Źródło: opracowanie własne.

Osoba	**Zakres obowiązków w zespole**
Piotr Minta (lider)	aplikacja do zarządzania wypożyczalnią, biblioteka elementów powszechnie używanych w aplikacjach, formularze, listy, paginacja, sortowanie, ustawienia
Michał Stasik	w aplikacji do zarządzania wypożyczalnią: moduły floty, klientów, finansów, wydruków
Michał Gołaszewski	aplikacja do korzystania z usług wypożyczalni, wsparcie w aplikacji do administrowania wypożyczalniami i aplikacji do prezentacji oferty systemu, obsługa subskrypcji
Maciej Alfut	aplikacja do administrowania wypożyczalniami, aplikacja do prezentacji oferty systemu, w aplikacji do zarządzania wypożyczalniami: dodatki, oddziały, oferty, rezerwacje, metody płatności, szablony wiadomości e-mail i dokumentów
Łukasz Lendzion	wsparcie początkowych prac przy aplikacji do prezentacji oferty systemu

3.3.7. Aplikacja mobilna

Aplikację mobilną realizował samodzielnie Łukasz Lendzion. Obejmowała ona procesy operacyjne pracownika wypożyczalni – wydanie i zwrot pojazdu, zgłaszanie usterek, profil użytkownika.

3.3.8. DevOps i infrastruktura

Obszar DevOps i infrastruktury prowadził samodzielnie Michał Gołaszewski. Obejmował on środowiska deweloperskie i produkcyjne, procesy ciągłej integracji i wdrażania, monitoring, rejestr obrazów kontenerowych oraz konfigurację domen i serwera *proxy*. Właściciele poszczególnych repozytoriów wspierali Michała punktowo w zakresie konfiguracji dotyczących własnych komponentów (pliki Dockerfile, manifesty wdrożeniowe, elementy *pipeline'ów*).

3.3.9. Testowanie

Zespół testerów koordynował działania kontrolne oraz nadzorował proces rejestracji i rozwiązywania błędów. Testy systemu odbywały się w miarę możliwości na bieżąco, równolegle z implementacją kolejnych funkcji, a w końcowej fazie projektu zostały przeprowadzone akceptacyjne testy manualne obejmujące cały system.

Tabela 7 Zakres obowiązków w zespole testowania. Źródło: opracowanie własne.

Osoba	**Zakres obowiązków w zespole**
Michał Stasik (lider)	testy automatyczne i manualne, organizacja procesu testowania
Łukasz Lendzion, Wojciech Matusiewicz, Maciej Sobolewski, Michał Gołaszewski, Michał Stasik	akceptacyjne testy manualne w końcowej fazie projektu

3.3.10. Integracja z systemem ERP

Integrację z systemem ERP prowadził samodzielnie Kacper Gajdamowicz. Obszar nie był elementem pierwotnego podziału na zespoły – został dodany w trakcie projektu jako rozszerzenie zakresu. Otrzymał odrębną ścieżkę realizacji, ponieważ dotyczył integracji systemu Wellhicle z zewnętrznym rozwiązaniem biznesowym (enova365).

3.3.11. Dokumentacja końcowa

Dokumentacja końcowa stała się pełnoprawnym strumieniem pracy w ostatniej fazie projektu, z dedykowanym projektem Jira i osobnym procesem recenzji. Treść dokumentu powstawała wspólnym wysiłkiem wszystkich uczestników – każdy członek zespołu opracowywał opisy własnych obszarów odpowiedzialności i recenzował rozdziały przygotowane przez pozostałych. Procesem kierował Wojciech Matusiewicz, który odpowiadał za organizację prac, formalizację procedur, przygotowanie poradników oraz konsolidację materiałów w spójny dokument końcowy. Kacper Gajdamowicz oraz Maciej Alfut wspierali koordynację zespołu.

3.4. Narzędzia pracy zespołowej

Do organizacji pracy wykorzystano kilka narzędzi, z których każde pełniło inną funkcję – wspierając planowanie, komunikację, dokumentowanie decyzji oraz pracę nad kodem i przygotowanie końcowej dokumentacji. W kolejnych podpunktach opisano sposób konfiguracji każdego z narzędzi w projekcie oraz ich wykorzystanie.

3.4.1. Jira

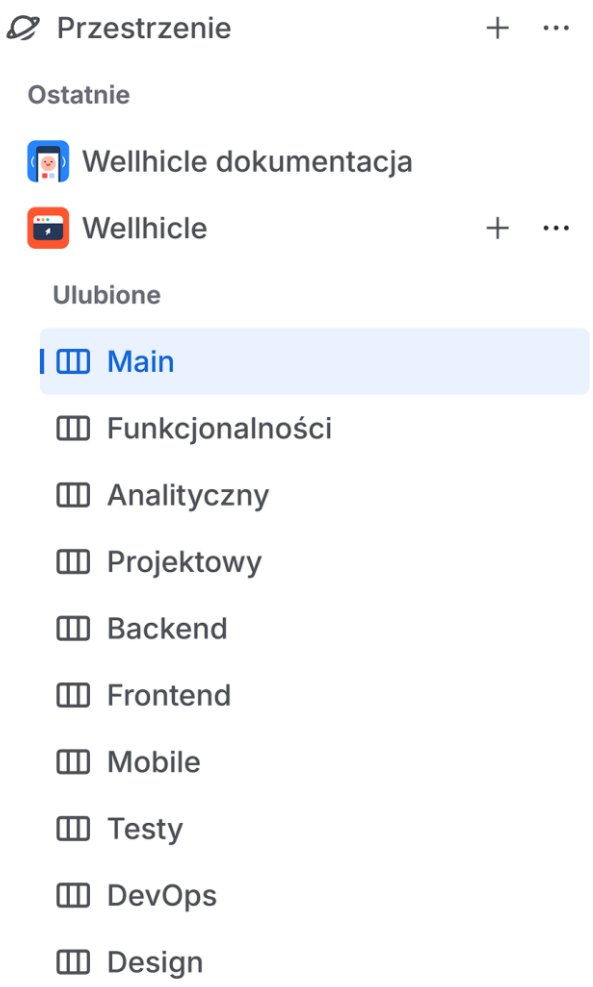
Jira [5] była podstawowym narzędziem organizacji pracy w projekcie Wellhicle. W Jirze rejestrowano zgłoszenia, ich statusy, priorytety oraz terminy realizacji. Było to również narzędzie służące do śledzenia postępów prac nad poszczególnymi funkcjami. Inne systemy opisane w pozostałych podpunktach pełniły rolę uzupełniającą: dokumentowały kontekst, kod źródłowy, bieżącą komunikację oraz materiały robocze. Jednak to Jira stanowiła centralne miejsce zarządzania planem pracy; pozwalała odtworzyć przebieg realizacji każdej funkcji i określić aktualny status projektu.

Wybór tego narzędzia wynikał z dwóch kluczowych czynników:

- chęci zapoznania się z rozwiązaniem powszechnie stosowanym w branży IT,
- po analizie alternatyw takich jak Trello uznano, że Jira znacznie lepiej sprawdza się w przypadku projektów angażujących wiele zespołów, różne typy zadań i skomplikowane zależności między nimi.

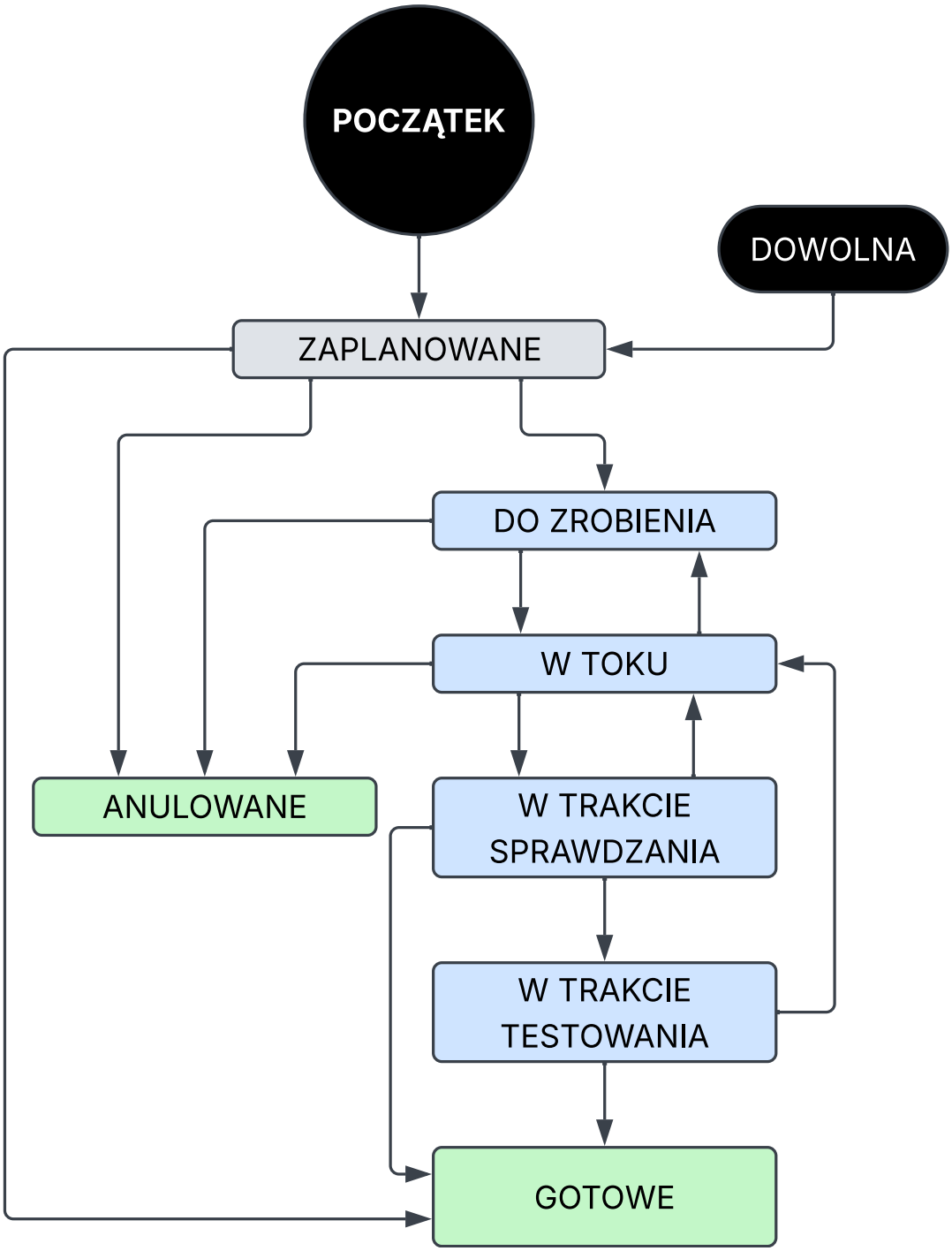
Główny projekt zarządzania pracą oznaczony kluczem „WELL” został skonfigurowany jako projekt typu *classic*, przeznaczony do pracy wielu zespołów. Początkowo utworzono go dla pojedynczego zespołu, jednak szybko okazało się to niewystarczające – taki wariant udostępniał wyłącznie jedną tablicę zadań i nie umożliwiał osobnego śledzenia prac w różnych obszarach. Projekt został ponownie utworzony, tym razem w konfiguracji wspierającej wiele zespołów. Wykorzystywano cztery typy zgłoszeń: epiki (duża jednostka pracy), zadania (ang. *tasks*), podzadania (ang. *subtasks*) oraz błędy (ang. *bugs*). Każdej tablicy przypisano pola priorytetu i terminu realizacji.

Dla każdego zespołu utworzono dedykowaną tablicę kanban, a w samych zgłoszeniach dodano pola pomocnicze: „Recenzent” oraz „Tester”, co ułatwiało przypisywanie odpowiedzialności na poszczególnych etapach procesu. Struktura projektu „WELL” obejmowała tablicę główną oraz osobne tablice dla każdego zespołu – schemat tej organizacji przedstawia *Rysunek 9.*



Rysunek 9 Struktura projektów Jira z podziałem na tablice zespołów. Źródło: opracowanie własne.

Aktualny przepływ statusów stosowany w projekcie „WELL" przedstawiono na *Rysunek 10*. Sekwencja rozpoczyna się od statusu „Zaplanowane", przechodzi przez „Do zrobienia" i „W toku", a następnie kończy się na etapach „W trakcie testowania" lub „W trakcie sprawdzania", by ostatecznie osiągnąć status „Gotowe". Status „Anulowane" służy do obsługi zgłoszeń, które nie wymagają dalszej realizacji – najczęściej są to duplikaty lub zadania utracone na skutek zmiany zakresu. Dodatkowa ścieżka „Dowolna" -> „Zaplanowane", umożliwiająca cofnięcie zgłoszenia do fazy planowania z dowolnego innego statusu, została wprowadzona w trakcie implementacji. Decyzja ta wynikała z obserwacji, że zakres zadań często ulegał przedefiniowaniu, a pierwotny jednokierunkowy przepływ utrudniał porządkowanie zgłoszeń wymagających ponownej analizy.



Rysunek 10 Przepływ statusów zgłoszeń w projekcie „WELL". Źródło: opracowanie własne.

Przepływ ten jest wynikiem stopniowego upraszczania pierwotnej konfiguracji. Początkowo zakładano siedem statusów: „Zaplanowane", „Do zrobienia", „W toku", „W trakcie sprawdzania", „W trakcie testowania", „W trakcie dokumentacji" oraz „Gotowe", co miało odzwierciedlać pełną idealną ścieżkę realizacji zadania. W praktyce model okazał się jednak zbyt restrykcyjny – nie każde zadanie wymagało przejścia przez wszystkie etapy, a część wracała do poprzednich statusów po uwzględnieniu uwag lub zmianie zakresu. Szczególnie problematyczny okazał się status „W trakcie dokumentacji", który pozostawiał otwarte zadania, mimo że ich część techniczna była już ukończona, co utrudniało ocenę rzeczywistego postępu. Status ten został wycofany, a sama końcowa dokumentacja w trakcie projektu uzyskała osobny projekt Jira „WD" z dedykowanymi statusami i procesem recenzji (szczegóły przedstawiono w podrozdziale *3.7*).

W codziennej pracy Jira służyła do planowania, kontroli zadań, nadawania priorytetu, komentowania oraz przypisywania odpowiedzialności. Narzędzie to było szczególnie przydatne w śledzeniu błędów – pozwalało nadać im priorytet, wyznaczyć osobę odpowiedzialną za ich rozwiązanie oraz przypisać testera. Dzięki temu proces zgłaszania i naprawiania usterek był efektywnie monitorowany.

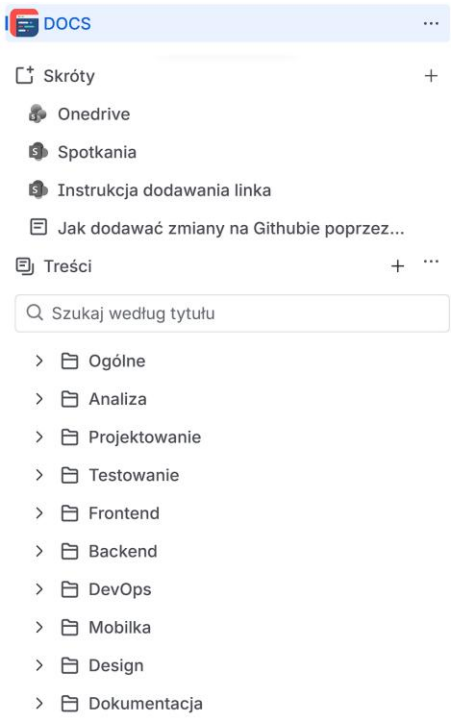
W trakcie realizacji projektu w Jirze zgromadziła się znaczna liczba zgłoszeń, co wynikało ze struktury organizacyjnej pracy. Po zamknięciu fazy analitycznej dla każdej funkcjonalności tworzono osobny epik, a do niego przypisywano komplet zadań w obszarach API, *designu* oraz widoków *webowych* i mobilnych. Taki układ sprawdzał się dobrze, ponieważ pozwalał na śledzenie postępów prac niezależnie dla każdego z tych zakresów. Wymagał jednak późniejszej korekty – konieczne było anulowanie części zadań zespołu *designu*, głównie tych przygotowanych wcześniej i ostatecznie anulowanych. Dodatkowo pojawiał się problem sytuacji, gdy funkcjonalność była technicznie obsługiwana wspólnym fragmentem implementacji (np. tym samym *backendowym* punktem końcowym dla panelu wypożyczalni i panelu administracyjnego), co rodziło pytania o zasadność prowadzenia osobnych zgłoszeń dla tych obszarów.

Niezależnie od struktury pracy, aktualność statusów była utrudniona przez realia projektu. Intensywność prac i ograniczone zasoby czasowe uczestników sprawiały, że bieżące aktualizowanie zgłoszeń traktowano jako dodatkowe obciążenie. Gdy statusy nie były odświeżane, system przestawał odzwierciedlać rzeczywisty stan prac. Mechanizmem walidującym faktyczny stan sytuacji były cykliczne spotkania statusowe, jednak brak aktualnego obrazu w Jirze w momencie ich trwania powodował konieczność prowadzenia rozmów na podstawie nieaktualnych danych oraz wymuszał doraźną aktualizację statusów podczas spotkań. W obu przypadkach wydłużało to czas trwania tych spotkań. Brakowało wyraźnie ustalonej zasady, że każdy uczestnik powinien aktualizować swoje zgłoszenia bezpośrednio przed rozpoczęciem spotkania.

3.4.2. Confluence

Confluence [6] pełniło w projekcie rolę przestrzeni dokumentacji powiązanej z Jirą. Wybór wynikał z obecności Confluence i Jiry w tym samym ekosystemie Atlassian – dzięki temu zadania zapisane w Jirze mogły bezpośrednio odnosić się do konkretnych zmian wprowadzanych w systemie, a dokumentacja Confluence służyła jako miejsce przeznaczone na szczegółowy kontekst techniczny, procedury i decyzje, które nie powinny być ograniczone wyłącznie do komentarzy przy zgłoszeniach.

W Confluence utworzono przestrzeń „DOCS”, gdzie przechowywano strony i materiały dla poszczególnych zespołów oraz dokumenty ogólne. Struktura tej przestrzeni, odzwierciedlająca podział obszarowy projektu, opisany w podrozdziale *3.3*, została przedstawiona na *Rysunek 11*. Znajdowały się tam m.in.: opisy techniczne, procedury operacyjne, instrukcje dotyczące procesów oraz wybrane ustalenia projektowe – np. integracja Jiry z Discordem, zasady przygotowania materiałów do dokumentu końcowego czy szczegółowa procedura testów manualnych. Confluence okazał się skutecznym narzędziem do przechowywania dłuższych treści niż standardowe komentarze w zgłoszeniach, umożliwiając uporządkowane przedstawienie procedur lub rozwiązań technicznych. Wybrane materiały zostały później wykorzystane jako pomocnicze źródło przy tworzeniu dokumentacji końcowej, ponieważ część decyzji była już udokumentowana.



Rysunek 11 Struktura przestrzeni dokumentacyjnej „DOCS" w Confluence. Źródło: opracowanie własne.

Zespół zakładał, że dokumentacja dla większych zadań powinna być systematycznie uzupełniana na platformie Confluence, jednak w praktyce ten mechanizm nie był konsekwentnie realizowany. Przy dużej intensywności prac i ograniczonych możliwościach czasowych członków zespołu implementacja rozwiązań oraz naprawa problemów postępowały szybciej niż tworzenie dokumentacji – bieżące opisywanie wymagałoby dodatkowego nakładu pracy, którego brakowało. W rezultacie robocza dokumentacja wspierała wybrane obszary, ale nie zastąpiła późniejszej fazy przygotowania ostatecznego dokumentu końcowego.

3.4.3. GitHub

GitHub [7] pełniło rolę podstawowego miejsca pracy nad kodem źródłowym. Wybór wynikał ze standardu branżowego oraz znajomości narzędzia przez członków zespołu, a struktura repozytoriów odpowiadała podziałowi systemu na komponenty.

Kod był przechowywany w organizacji GitHub. Dostępy zostały zorganizowane przez zespoły, takie jak Backend, DevOps, Frontend i Mobile, według zasady, że wszyscy uczestnicy mają uprawnienia do odczytu repozytoriów, natomiast możliwość wprowadzania zmian jest uzależniona od przynależności do danego zespołu i zakresu jego obowiązków. Repozytoria zostały rozdzielone na komponenty systemu: serwer aplikacyjny, cztery aplikacje *frontendowe* i aplikacja mobilna, wspólną bibliotekę *frontendową*, samodzielną aplikację mobilną oraz infrastrukturę techniczną. Praca w GitHubie była ściśle powiązana z Jirą – nazwy gałęzi rozpoczynano od identyfikatora zgłoszenia (np. „WELL-123"), co umożliwiało bezpośrednie kojarzenie zmian w kodzie z konkretnymi zadaniami. *Pull requesty* musiały zawierać opis zakresu modyfikacji, odwołanie do powiązanego zgłoszenia oraz umożliwiać przeprowadzenie recenzji przed scaleniem zmian. W repozytoriach przygotowano szablony dokumentacji *pull requestów*, a w Confluence udostępniono instrukcję dotyczącą procedury zgłaszania i recenzowania zmian za pomocą tego narzędzia.

GitHub dobrze wspierał równoległą pracę nad komponentami. Historia zmian, *pull requesty* i powiązania z zadaniami pomagały odtwarzać kontekst implementacji, a wewnętrzne pakiety – biblioteka elementów powszechnie używanych w aplikacjach „@wellhicle/common" po stronie

frontendu oraz pakiet klienta API „Wellhicle.Client” po stronie integracji – ułatwiały współdzielenie kodu między komponentami. Ograniczeniem był brak płatnej organizacji, który utrudniał formalną ochronę głównych gałęzi (np. wymuszanie recenzji przed scaleniem). Część tych zasad trzeba było zastąpić ustaleniami organizacyjnymi: pracą na osobnych gałęziach, pilnowaniem *opisów pull requestów* i ręcznym podłączaniem zgłoszeń Jiry.

3.4.4. Microsoft Teams

Microsoft Teams [8] był wykorzystywany przede wszystkim do spotkań oficjalnych i nagrywanych. Jego znaczenie wynikało nie tylko z możliwości prowadzenia wideokonferencji, lecz głównie z dostępności nagrań, transkrypcji oraz podsumowań spotkań – funkcji kluczowych w projekcie o dużej liczbie rozmów organizacyjnych i technicznych. Teams był wybierany tam, gdzie istotne było zachowanie trwałego śladu ustaleń, dotyczyło to szczególnie cyklicznych spotkań statusowych. Podczas tych sesji wykorzystywano funkcje dostępne w planie Teams Premium, takie jak automatycznie generowane podsumowania i notatki. Dzięki temu osoby nieobecne mogły szybko zapoznać się z kluczowymi decyzjami, a cały zespół budował łatwo dostępną bazę wiedzy do późniejszego przeszukiwania.

Ograniczeniem Teams było to, że funkcje premium zależały od jednego konta – nie każdy członek zespołu mógł samodzielnie organizować spotkań z pełnym zestawem funkcji nagrywania i podsumowań. Część osób preferowała również Discord, szczególnie przy szybkich spotkaniach technicznych. W rezultacie dużo rozmów nie zostało udokumentowanych w takim samym stopniu.

3.4.5. Discord

Discord [9] był podstawowym kanałem codziennej komunikacji zespołu. Wybór narzędzia wynikał z jego wygody, szybkości wymiany informacji oraz znajomości platformy przez członków projektu. Rozważano również Slacka, jednak darmowa wersja tego rozwiązania nie oferowała funkcji prowadzenia spotkań grupowych na potrzeby zespołu.

Serwer Discord został uporządkowany poprzez utworzenie kanałów tematycznych i ról członkowskich. Kanały te były powiązane z obszarami pracy oraz komunikacją bieżącą, a role serwera odpowiadały poszczególnym zespołom, umożliwiając kierowanie powiadomień do właściwych osób. Wyodrębniono również dedykowany kanał, przeznaczony wyłącznie na powiadomienia ze zgłoszeń – konfigurację tego kanału oraz towarzyszącego mu bota opisano w punkcie *3.4.7*.

Największą zaletą Discorda była szybkość komunikacji. Narzędzie sprawdzało się przy krótkich pytaniach, wstępnej walidacji problemów oraz konsultacjach technicznych między spotkaniami statusowymi. Jego główną słabością była jednak archiwizacja decyzji – choć przyspieszało rozwiązanie bieżących zagadnień, ważne ustalenia wymagały późniejszej dokumentacji (np. w Jirze lub Confluence), aby mogły być odtworzone po czasie.

3.4.6. OneDrive

OneDrive [10] służyło do przechowywania plików projektowych, dokumentów roboczych, arkuszy kalkulacyjnych, nagrań oraz materiałów analitycznych i końcowych. Był niezbędny w przypadku dużych zbiorów danych lub formatów (np. pliki Word, Excel), które Confluence nie obsługiwało wygodnie: przy obszerniejszych dokumentach tekstowych i arkuszach kalkulacyjnych, nagraniach wideo czy materiałach wymagających komentarzy.

Początkowo zespół korzystał z Google Drive jako miejsca na nagrania spotkań i większe pliki. W trakcie projektu podjęto decyzję o przejściu na OneDrive, co wynikało z dwóch głównych powodów:

- aktywne linki do zasobów Google Drive osadzane w Confluence nie działały poprawnie ze względu na ograniczenia uprawnień w domenie uczelnianej,
- dokument końcowy pracy inżynierskiej miał być przygotowany w formacie Microsoft Word, a edytor Google Docs nie zapewniał wystarczających możliwości współpracy z tym formatem.

OneDrive, dzięki natywnej integracji z pakietem Microsoft Office, okazał się jedynym rozwiązaniem odpowiednim dla materiałów przeznaczonych do ostatecznego dokumentu. W końcowej fazie projektu OneDrive pełniło funkcję głównego repozytorium: gromadzono tam pliki źródłowe i robocze, scenariusze testowe, arkusze analityczne oraz dokumenty przygotowywane do przeniesienia do wersji końcowej.

3.4.7. Bot powiadomień z Jiry

Bot powiadomień z Jiry był pomocniczym narzędziem organizacji pracy zespołu, wdrożonym w celu uporządkowania komunikacji między systemem Jira, a serwerem Discord wykorzystywanym przez członków projektu Wellhicle. Rozwiązanie miało ograniczyć konieczność śledzenia jednego wspólnego kanału z powiadomieniami przez wszystkie osoby uczestniczące w projekcie.

Mechanizm działania opierał się na natywnej automatyzacji skonfigurowanej po stronie Jiry oraz wbudowanej funkcjonalności Discorda, jaką jest wystawianie *webhooków*. Ograniczeniem *webhooka* był fakt, że może on publikować wiadomości tylko na jednym kanale, co uniemożliwiało przekazywanie powiadomień do kanałów odpowiednich zespołów. Z uwagi na to ograniczenie powstał bot rozszerzający funkcjonalność Discorda. Jira decydowała, które zdarzenia powinny zostać wysłane jako powiadomienia. W treści takiej wiadomości znajdowało się oznaczenie roli Discord odpowiadającej danemu zespołowi. Bot, napisany w Pythonie z użyciem biblioteki discord.py [11] i uruchamiany w wirtualnym środowisku, nasłuchiwał wiadomości publikowanych na kanale technicznym połączonym z *webhookiem*.

Po wykryciu oznaczenia roli, bot przekazywał otrzymaną wiadomość na kanał właściwego zespołu. Przypisanie zespołów do kanałów było realizowane na podstawie mapowania identyfikatora roli na identyfikator kanału. Bot nie filtrował samodzielnie typów zdarzeń z Jiry, lecz rozdzielał dalej wszystkie powiadomienia, które zostały przesłane przez skonfigurowaną automatyzację.

Rozwiązanie wspierało organizację komunikacji wokół pracy w Jirze i Discordzie, szczególnie w sytuacji równoległej pracy kilku zespołów. Jego ograniczeniem była jednak zależność od płatnych mechanizmów automatyzacji Atlassiana. Z bota przestano korzystać z powodu braku budżetu na dalsze wykorzystanie automatyzacji Jira. Kontynuowanie tej funkcji wymagałoby płatnej subskrypcji.

3.5. Komunikacja i spotkania

Komunikacja w projekcie miała charakter wielokanałowy – wykorzystywano kilka narzędzi szczegółowo opisanych w podrozdziale *3.4*. W niniejszej części uzupełniono tamten opis o logikę podziału kanałów oraz organizację spotkań, których rytm i typy wpływały na sposób koordynacji prac między obszarami.

Każdy z kanałów pełnił rolę dopasowaną do charakteru ustalenia. Logikę podziału przedstawiono w *Tabela 8*.

Tabela 8 Kanały komunikacji i ich typowe zastosowanie w projekcie. Źródło: opracowanie własne.

Kanał	**Typowe zastosowanie**
Discord	bieżąca komunikacja, krótkie pytania, wstępna walidacja problemów
Jira	zadania, statusy, błędy, decyzje wymagające trwałego śladu
Confluence	procedury techniczne, ustalenia projektowe, dokumentacja robocza
GitHub	praca z kodem (gałęzie, pull requesty, recenzje)
OneDrive	dokumenty Word i Excel, nagrania, materiały do dokumentu końcowego
Microsoft Teams	spotkania oficjalne i nagrywane

Harmonogram spotkań statusowych ustalono na każdą środę o godzinie 20:00 oraz każdą niedzielę, na którą nie przypadał zjazd na uczelni, o 12:00 (zob. podrozdział *3.2*). Spotkania w środę umożliwiały zaplanowanie prac do końca tygodnia, natomiast te w niedzielę pozwalały na podsumowanie wykonanych działań i przygotowanie kolejnych kroków.

Indeks spotkań projektu obejmuje wyłącznie te, których przebieg został utrwalony. W praktyce były to niemal wszystkie spotkania prowadzone za pośrednictwem Microsoft Teams, gdzie rejestracja ustaleń odbywała się automatycznie. Krótkie konsultacje na innych platformach (np. Discord) z reguły nie były dokumentowane i do indeksu ich nie włączono. Do połowy maja 2026 r. odnotowano 78 spotkań, przy czym lista pozostaje otwarta ze względu na ciągłe prowadzenie kolejnych. Zarejestrowane spotkania można podzielić na pięć kategorii: **statusowe** (43 spotkania), **tematyczne** (18 spotkań poświęconych konkretnym zagadnieniom, takim jak wymogi językowe czy obsługa finansów), **analityczne** (10 spotkań dotyczących analizy wymagań i scenariuszy użytkowania), **projektowe** (3 sesje decyzyjne związane z architekturą systemu) oraz **pozostałe** (3 inne spotkania). Analiza rozkładu pokazuje, że podstawę koordynacji stanowiły regularne spotkania statusowe, jednak w miarę postępów projektowych pojawiała się również potrzeba dedykowanych sesji tematycznych – szczególnie gdy nowe zagadnienia lub zmiany zakresu wymagały pogłębionej dyskusji w mniejszym gronie.

Niezależnie od opisanych powyżej kanałów uczestnicy projektu utrzymywali kontakty bezpośrednie – rozmowy telefoniczne, wiadomości tekstowe oraz komunikatory osobiste. Wykorzystywano je głównie wtedy, gdy temat dotyczył wąskiego grona osób i nie wymagał organizowania odrębnego spotkania ani angażowania całego zespołu.

3.6. Zmiany zakresu projektu

Zakres projektu nie był stały przez cały okres realizacji – ewoluował zarówno w kierunku rozszerzania, jak i ograniczania. Część zmian wynikała z pogłębionej analizy domeny w trakcie implementacji, część zaś była świadomą decyzją redukcji zakresu ze względu na ograniczenia czasowe i jakościowe. Kluczową zmianą okazał się rozwój modułu finansowego oraz jego integracja z zewnętrznym systemem ERP, co zostało szczegółowo opisane w punkcie *3.6.1*, pozostałe modyfikacje ujęto łącznie w punkcie *3.6.2.*

3.6.1. Moduł finansowy oraz integracja z systemem ERP

Decyzja o rozszerzeniu zakresu projektu o moduł finansowy wraz z integracją zewnętrznym systemem ERP została podjęta pod koniec stycznia 2026 r., jako naturalna kontynuacja prac nad funkcjami finansowymi platformy. Moduł ten zaprojektowano jako warstwę odpowiedzialną za obsługę zdarzeń gospodarczych w systemie, obejmującą zarówno rezerwacje po stronie wypożyczalni (transakcje z klientem wypożyczalni), jak i subskrypcje platformy (sprzedaż dostępu dla właścicieli). Założono, że formalne rozliczenia tych zdarzeń – wystawianie faktur oraz ich ewidencja księgowa –

zostaną przekazane zewnętrznemu systemowi ERP (w tym przypadku enova365), co pozwoliło uniknąć samodzielnej implementacji warstwy fakturowania w ramach platformy. Integracja umożliwiła automatyczne wystawianie i odsyłanie dokumentów do systemu Wellhicle. W obrębie modułu finansowego zaimplementowano m.in.: obsługę subskrypcji wraz z mechanizmem ich automatycznego odnawiania, możliwość realizacji zwrotów (pełnych i częściowych) oraz funkcję pozwalającą na podgląd rachunku przed finalnym utworzeniem. Wprowadzenie modułu znacząco poszerzyło zakres prac implementacyjnych, a integracja z ERP stanowiła kluczowe dopełnienie funkcjonalności rozliczeniowej po stronie platformy. Szczegóły techniczne tej części systemu zostały omówione w podrozdziale *7.5*

3.6.2. Pozostałe zmiany zakresu

Pozostałe rozszerzenia zakresu wynikały głównie z pogłębienia analizy w trakcie implementacji. Początkowo system miał być wielojęzyczny, z utrzymaniem jednej waluty oraz strefy czasowej. W toku prac obsługa międzynarodowa została rozszerzona o wielowalutowość i operowanie różnymi strefami czasowymi, co wymagało modyfikacji zarówno po stronie modelu danych, jak i interfejsu użytkownika. Wprowadzono także nowe elementy do modelu domenowego: grupy pojazdów (umożliwiające wspólną konfigurację cen i zasad dla podobnych typów), manualne blokady dostępności pozwalające wyłączyć pojazd bez tworzenia sztucznej rezerwacji oraz konfigurowalną jednostkę przebiegu (kilometry lub mile) ustawianą na poziomie oddziału.

Po stronie świadomych redukcji zakresu wycofano funkcję umożliwiającą właścicielowi wypożyczalni wylogowanie pracownika. Funkcja ta okazała się znacznie bardziej kosztowna w implementacji, niż wynikało to z jej pierwotnego opisu – rzetelna realizacja wymagałaby wprowadzenia modyfikacji, której budowa wykraczała poza zaplanowany zakres prac.

Po opisanych zmianach zakres projektu ustabilizował się w końcowej fazie realizacji – priorytetem stało się zakończenie implementacji zaplanowanych funkcji oraz przygotowanie dokumentacji końcowej, a kolejne propozycje rozszerzeń były świadomie odkładane na dalsze etapy rozwoju produktu.

3.7. Organizacja dokumentacji końcowej

W celu przygotowania dokumentacji końcowej opracowano i udostępniono zespołowi w Confuence zbiorcze zasady dotyczące tworzenia fragmentów dokumentacji. Zasady te obejmowały organizację pracy nad draftami, kryteria uznania materiału za gotowy do scalenia oraz sposób przekazywania uwag, a także praktyczne reguły redagowania tekstu, obsługi komentarzy, przygotowywania rysunków, tabel, listingów i cytowań. Dzięki przyjętym rozwiązaniom proces tworzenia dokumentacji końcowej zyskał bardziej uporządkowany i jednolity charakter.

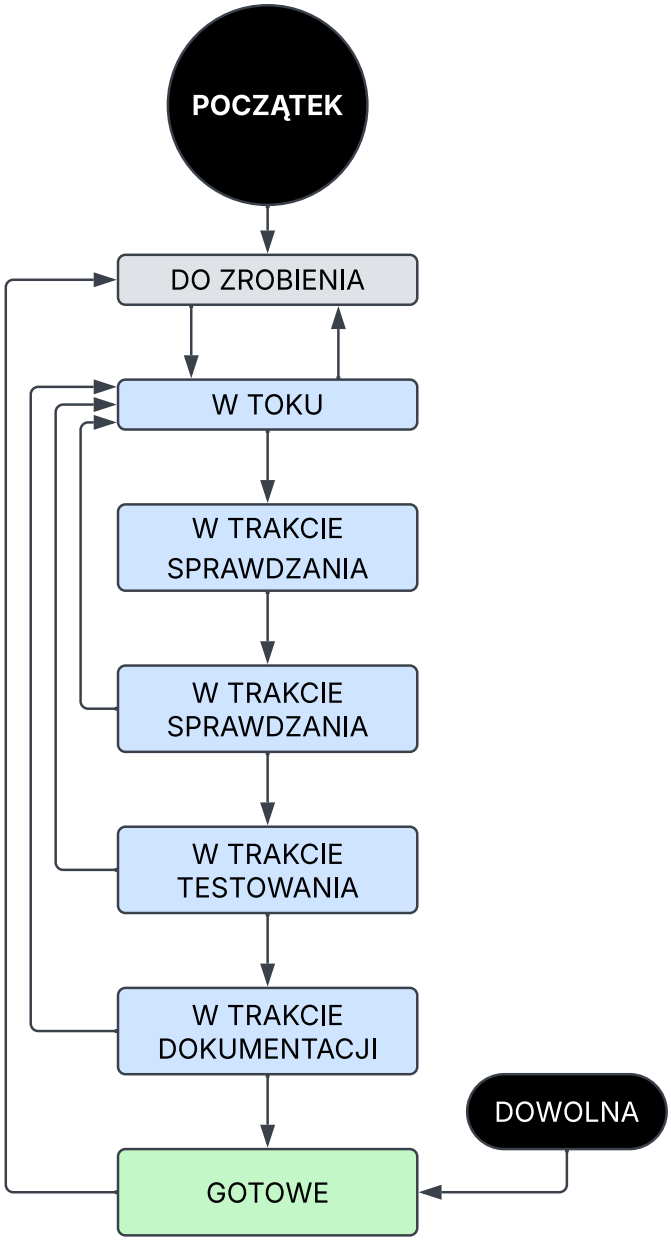
3.7.1. Nowy projekt w Jira

W końcowym etapie implementacji utworzono w Jira nowy projekt "WD" przeznaczony wyłącznie na organizację przygotowania dokumentacji końcowej. W ramach projektu „WD” utworzono zgłoszenia odpowiadające poszczególnym rozdziałom, które następnie przypisano osobom merytorycznie odpowiedzialnym za dane obszary. Odpowiedzialność za zgłoszenie rozdziałowe oznaczała konieczność doprowadzenia treści do stanu umożliwiającego recenzję i scalenie z dokumentem głównym. W przypadku delegacji zadań tworzono podzadania w ramach danego zgłoszenia, co pozwalało na śledzenie postępów i przejrzystość procesu.

Przepływ pracy nad fragmentami dokumentacji przedstawiony na *Rysunek 12* został powiązany ze statusami

- „W toku”,
- „W trakcie sprawdzania”,
- „W trakcie testowania”,
- „W trakcie dokumentacji”.

Status „W toku” oznaczał przygotowanie lub poprawianie treści, natomiast „W trakcie sprawdzania” obejmował recenzję wykonywaną przez osobę z tego samego obszaru projektu. Po jej zakończeniu tekst przechodził etap weryfikacji przez osobę niezwiązaną bezpośrednio z opisywanym tematem – status „W trakcie testowania”, co pozwalało sprawdzić łatwość zrozumienia fragmentu dla czytelnika spoza danego obszaru. Status „W trakcie dokumentacji” oznaczał, że fragment dokumentu jest gotowy do włączenia go do dokumentu głównego.



Rysunek 12 Przepływ statusów zgłoszeń w projekcie „WD". Źródło: opracowanie własne.

3.7.2. Obieg dokumentu

Równolegle uporządkowano obieg dokumentów roboczych na OneDrive. Folder “Drafty materiałów do dokumentu końcowego” służył do przechowywania wersji roboczych, materiałów w recenzji oraz fragmentów zwróconych do poprawek. Po zakończeniu recenzji plik przenoszono do folderu “Materiały do dodania do dokumentu końcowego”, co oznaczało, że może zostać rozpatrzony przy scalaniu z dokumentem głównym. Po zaakceptowaniu i włączeniu treści do pracy źródłowy plik trafiał do folderu “Materiały dodane do dokumentu końcowego”, pełniącego funkcję archiwum materiałów już wykorzystanych.

Dodatkowym elementem kontroli było ujednolicenie zasad przygotowywania plików. Materiały nazywano według schematu obejmującego numer sekcji oraz nazwisko autora, co ułatwiało przypisanie fragmentu do struktury dokumentu i identyfikację osoby odpowiedzialnej za jego przygotowanie. Uwagi przekazywano głównie w komentarzach w Wordzie, bez możliwości wprowadzenia zmian w dokumencie roboczym. Dokument główny edytował wyłącznie Wojciech Matusiewicz, który scalał zaakceptowane materiały i dbał o ich spójność redakcyjną. Przyjęcie takiego rozwiązania pozwoliło na jasne wydzielenie odpowiedzialności za poprawność dokumentu.

Wydzielenie osobnego procesu dla dokumentacji końcowej pozwoliło oddzielić pracę redakcyjną od bieżących zadań technicznych i ograniczyć narastanie długu dokumentacyjnego. Przyjęta organizacja pracy sprawiła zarazem, że opracowanie końcowej wersji dokumentu opierało się na scalaniu informacji pochodzących z różnych źródeł, w tym z zadań Jira, materiałów OneDrive, ustaleń ze spotkań i dokumentów roboczych.

3.8. Korekty planu i wnioski organizacyjne

Plan pierwotny, zapisany w konspekcie pracy, dzielił realizację projektu na jedenaście faz – od fazy wstępnej (przygotowanie i uczenie się) przez analizę wymagań i opracowanie koncepcji systemu, projekt architektury oraz bazy danych, projekt interfejsu użytkownika (UI/UX), dwie fazy implementacji (backend i frontend), testy integracyjne z poprawkami, moduły rozszerzone wraz z DevOps, aż po przygotowanie dokumentacji technicznej oraz prezentację końcową. W praktyce realizacja projektu odbiegała od tego modelu pod względem kolejności prac i ich zakresu czasowego. Korekty były świadome, wynikały z przyczyn systemowych opisanych szczegółowo w podrozdziałach *3.6* i *3.7.*

3.8.1. Korekty planu i przebiegu prac

Pierwotny układ faz z konspektu zakładał silne rozdzielenie czasowe poszczególnych etapów – analizy, projektowania, implementacji, testowania oraz dokumentacji. W praktyce taki podział okazał się trudny do utrzymania przy skali projektu i liczbie aplikacji składowych. Analiza była pogłębiana w trakcie implementacji, gdy pojawiały się konkretne zagadnienia domenowe nierozstrzygnięte na etapie zbierania wymagań. Projekt architektury systemu został zamknięty planowo podczas projektowania, a późniejsze rozszerzenia zakresu dodawano jako odrębne moduły spójne z istniejącą strukturą. Testy prowadzono na bieżąco równolegle z implementacją kolejnych funkcji, a nie tylko w wyznaczonym etapie testów integracyjnych. W styczniu 2026 r. zaostrzono nacisk na precyzyjne terminy realizacji ustalane podczas spotkań statusowych, natomiast od lutego 2026 r. świadomie ograniczano przyjmowanie nowych funkcji do zakresu projektu – jednocześnie rozszerzając go o moduł finansowy i integrację z systemem ERP (szczegóły w podrozdziale *3.6*). W kwietniu 2026 r. wydzielono dokumentację końcową jako osobny strumień pracy, wyposażony we własne zarządzanie Jira (projekt „WD”) i proces recenzji (szczegóły w podrozdziale *3.7*).

Tabela 9 Porównanie planu pierwotnego z rzeczywistym przebiegiem realizacji. Źródło: opracowanie własne.

Faza projektu (wg konspektu pracy)	**Okres realizacji (plan pierwotny)**	**Rzeczywisty przebieg**
1. Faza wstępna – przygotowanie i uczenie się	2025-10	Zgodnie z planem; podział ról i konfiguracja narzędzi opisane w podrozdziałach 3.3 i 3.4.
2. Analiza wymagań i opracowanie koncepcji systemu	2025-10 - 2025-11	Październik 2025 – grudzień 2025; pogłębiana również w wybranych obszarach podczas implementacji.
3. Projekt architektury i bazy danych	2025-11	Zrealizowany zgodnie z planem; późniejsze rozszerzenia zakresu projektu dodawane jako odrębne moduły (zob. podrozdział *3.6*).
4. Projekt interfejsu użytkownika (UI/UX)	2025-11 - 2025-12	Zrealizowany zgodnie z planem.
5. Implementacja -etap I (backend)	2025-12 - 2026-01	Rozpoczęta zgodnie z planem; zakończona w marcu-kwietniu 2026 r. Skala prac okazała się większa niż przewidywano, dodatkowo uwzględniając rozszerzenia (podrozdział *3.6*).
6. Implementacja – etap II (frontend)	2025-12 - 2026-01	Rozpoczęta zgodnie z planem; prowadzona równolegle do backendu; zakończona w marcu-kwietniu 2026 r.
7. Testy integracyjne i poprawki	2026-02 - 2026-03	Prowadzone na bieżąco od stycznia 2026 r.; testy akceptacyjne przeprowadzone pod koniec projektu.
8. Moduły rozszerzone i DevOps	2025-11 - 2026-01	Wdrażane równolegle do implementacji; moduły dodawano sukcesywnie (podrozdział *3.6*).
9. Dokumentacja projektu – część teoretyczna	2026-02 - 2026-04	Rozpoczęta z opóźnieniem; wydzielona w kwietniu 2026 r. jako osobny strumień (projekt Jira „WD”, podrozdział *3.7*).
10. Dokumentacja projektu – opis implementacji i testów	2026-04 - 2026-05	Prowadzona równolegle z częścią teoretyczną od kwietnia 2026 r.
11. Przygotowanie prezentacji i obrony	2026-05 - 2026-06	Zgodnie z planem.

3.8.2. Wnioski organizacyjne

Jednym z głównych ograniczeń organizacyjnych projektu była limitowana dostępność czasowa członków zespołu. Projekt realizowany był równolegle z innymi obowiązkami, takimi jak praca zawodowa, studia w trybie weekendowym oraz indywidualne zobowiązania poszczególnych uczestników. Praktyczną konsekwencją tego stanu rzeczy był ograniczony czas dostępny na działania wykraczające poza główny zakres implementacji, takie jak bieżąca aktualizacja dokumentacji w Jirze, zapisywanie ustaleń w Confluence czy utrzymanie spójności materiałów projektowych. Te zadania traktowane były jako dodatkowe obciążenia dla członków zespołu. Również dotrzymywanie terminów okazało się wyzwaniem – terminy ustalane podczas spotkań statusowych często przesuwano, a w niektórych ścieżkach projektu kolejne decyzje zastępowały wcześniejsze deklaracje.

Rola kierownictwa projektu była kluczowa i obejmowała istotny wkład merytoryczny w różnych aspektach prac. To połączenie funkcji okazało się znaczące, stanowiąc dodatkowe obciążenie ponad już intensywnymi działaniami projektowymi. W przypadku tak dużej inicjatywy korzystne byłoby wyodrębnienie dedykowanej roli menedżera projektu, oddzielonego od kierownictwa merytorycznego i skupionego na prowadzeniu spotkań, monitorowaniu terminów, kontroli zadań w Jirze. Menedżer ten koordynowałby również relacje między poszczególnymi obszarami projektu.

Alternatywą lub uzupełnieniem mogło być szersze wykorzystanie automatyzacji w zarządzaniu pracami i koordynowaniu działań zespołu, np.: powiązanie ustaleń ze spotkań z zadaniami czy monitorowanie statusu zgłoszeń. Narzędzia oparte na sztucznej inteligencji mogłyby wspierać wykrywanie blokad i śledzenie postępów, co odciążyłoby członków projektu w podejmowaniu decyzji organizacyjnych.

Osobnym wyzwaniem była organizacja dokumentacji projektowej. Status „W trakcie dokumentacji" w przepływie pracy Jiry, zgodnie z punktem *3.4.1*, miał na celu zapobieganie zamykaniu zadań technicznych przed udokumentowaniem ich zakresu. W praktyce jednak, ze względu na ograniczone możliwości czasowe uczestników projektu, prowadził do narastania długu dokumentacyjnego, który musiał być nadrabiany pod koniec prac jako osobny strumień (zgodnie z podrozdziałem *3.7*). Wnioskiem systemowym jest to, że w projektach tej skali należy wydzielić dokumentację od samego początku – poprzez utworzenie dedykowanego projektu Jira, z osobnym harmonogramem recenzji i wyznaczoną osobą prowadzącą proces. Dzięki temu uniknie się sytuacji, w której dług dokumentacyjny kumuluje się do momentu zamknięcia prac funkcjonalnych, konkurując o czas zespołu.

4. Narzędzia i technologie

W rozdziale przedstawiono narzędzia, biblioteki oraz technologie wykorzystane podczas prac nad systemem Wellhicle. Omówiono rozwiązania wybrane przez zespół, ich alternatywy i przesłanki wpływające na ostateczny dobór stosu narzędziowego oraz technologicznego.

4.1. Analiza

Do opracowania części analitycznej projektu wykorzystano narzędzia służące do tworzenia i organizowania diagramów. Część przygotowanych modeli miała charakter roboczy, natomiast wybrane diagramy zostały następnie wykorzystane w dokumentacji systemu Wellhicle.

4.1.1. Lucidchart

Lucidchart jest narzędziem do tworzenia diagramów, map procesów i innych wizualnych modeli. Udostępnia funkcje współpracy w czasie rzeczywistym, co pozwala kilku osobom pracować nad tym samym dokumentem projektowym [12].

W projekcie Wellhicle narzędzie Lucidchart było głównym oprogramowaniem wykorzystywanym do tworzenia diagramów analitycznych. Wszystkie diagramy powstałe w procesie analizy zostały umieszczone w jednym wspólnym dokumencie projektu, podzielonym na osobne strony odpowiadające konkretnym artefaktom. Dzięki temu materiały były uporządkowane i dostępne w jednej przestrzeni.

Przy skali tego projektu narzędzie zaczęło jednak wykazywać ograniczenia. Wraz ze wzrostem liczby stron w pojedynczym dokumencie zauważalnie spadała wydajność edytora, pojawiały się też sporadyczne błędy wyświetlania (m.in. nadmiarowe ciągi znaków w nazwach elementów), a brak wbudowanej kontroli wersji utrudniał śledzenie zmian w rozrastającym się zbiorze diagramów.

4.1.2. Draw.io

Draw.io to otwartoźródłowe oprogramowanie służące do tworzenia diagramów i schematów. Umożliwia projektowanie m.in. diagramów UML, map procesów oraz prostych modeli strukturalnych [13].

W projekcie Wellhicle narzędzie Draw.io wykorzystano na początkowym etapie modelowania diagramów przypadków użycia, głównie ze względu na możliwość tworzenia ich bezpośrednio w Confluence, co ułatwiało integrację dokumentacji opisowej z elementami graficznymi – do tego rodzaju diagramów narzędzie sprawdzało się dobrze. Próba zastosowania go do diagramów klas UML wypadła natomiast niekorzystnie: sposób modelowania okazał się na tyle niewygodny, że zrezygnowano z niego już po naszkicowaniu kilku klas. Z tego powodu dalsze modelowanie przeniesiono do Lucidchart, a rolę Draw.io ograniczono do wczesnego etapu analizy.

4.1.3. Microsoft Visio

Microsoft Visio to narzędzie firmy Microsoft służące do tworzenia diagramów, w tym schematów blokowych i diagramów procesów oraz innych modeli wizualnych wspierających analizę i dokumentowanie rozwiązań [14].

W projekcie Microsoft Visio wykorzystano pomocniczo oraz w celach badawczych – przy jego użyciu przygotowano roboczy szkic dotyczący modułu finansowego, który nie wszedł do finalnej dokumentacji, lecz pomógł doprecyzować założenia tego fragmentu systemu. Narzędzie oceniono jednak jako słabo przystosowane do modelowania UML: interfejs oraz sposób obsługi, w szczególności tworzenie i prowadzenie asocjacji, były niewygodne, przez co Visio nie nadawało się do pracy nad diagramami klas i pozostało jedynie narzędziem pomocniczym.

4.1.4. Enterprise Architect

Enterprise Architect firmy Sparx Systems to rozbudowane środowisko do modelowania w języku UML. W projekcie posłużył do opracowania finalnych diagramów klas: zarówno analitycznego modelu dziedziny (ACD, podrozdział *5.7*), jak i projektowego diagramu klas (PCD, podrozdział *6.3*). Ponieważ oba diagramy powstały w tym samym narzędziu, jego opis zebrano w jednym miejscu, poniżej [15].

Model został utworzony metodą inżynierii wstecznej poprzez import kodu warstwy domenowej serwera aplikacyjnego (punkt *7.1.2*) do narzędzia Enterprise Architect, co umożliwiło automatyczne załadowanie klas wraz z ich atrybutami i metodami. Surowy kod nie nadawał się jednak do bezpośredniego importu, ponieważ jest implementacją opartą na *frameworku* mapowania obiektowo-relacyjnego (Entity Framework Core) oraz na mechanizmach języka C#. Enterprise Architect podczas inżynierii wstecznej odwzorowuje konstrukcje kodu niemal jeden do jednego, przez co bez przygotowania, na diagramie znalazłyby się elementy nieistotne dla modelu pojęciowego UML: akcesory właściwości (`get`/`set`) wraz z inicjalizatorami, konstruktory, atrybuty mapowania i walidacji (np. `[Required]`, `[MaxLength]`), dyrektywy `using`, prywatne pola pomocnicze, pola audytowe (`CreatedAt`, `UpdatedAt`) oraz techniczne metody pomocnicze. Dlatego przed importem wykonano roboczą kopię projektu i jednorazowo usunięto z niej te elementy. Właściwości sprowadzono do zwykłych atrybutów klas, a spośród metod pozostawiono jedynie istotne operacje domenowe. Dzięki temu do narzędzia trafił czytelny model: klasy domenowe z atrybutami i kluczowymi operacjami. Asocjacje odwzorowały się podczas importu tylko częściowo: pojawiły się zdublowane powiązania skierowane, bez określonej liczności, dlatego relacje między klasami (ich liczność, agregacje, ograniczenia i komentarze), a także układ oraz kompozycję diagramów opracowano ręcznie. Diagram analityczny wymagał dodatkowo usunięcia elementów technicznych oraz spolszczenia nazewnictwa. Proces ten został szczegółowo opisany w rozdziale dotyczącym zastosowania sztucznej inteligencji.

Po Enterprise Architect sięgnięto, gdy wcześniejsze diagramy klas przestały w pełni odzwierciedlać finalnie zbudowany system po rozszerzeniu zakresu projektu. Przemodelowanie na podstawie aktualnego kodu pozwoliło uzyskać diagramy zgodne ze stanem rzeczywistym i to one stanowią w pracy wersję obowiązującą.

Narzędzie wyróżniało się obiektowym podejściem do modelowania: element raz utworzony jako obiekt, wraz z jego cechami i powiązaniami, jest następnie spójnie wykorzystywany w całym modelu, co bardzo ułatwiało pracę nad diagramem klas. Doceniono również wbudowaną kontrolę wersji oraz wygodny eksport diagramów, w tym do formatów wektorowych. Wejście w narzędzie wymagało czasu – interfejs jest rozbudowany i oferuje bardzo wiele opcji, jednak po jego opanowaniu Enterprise Architect okazał się spośród rozważanych narzędzi najlepiej dopasowany do modelowania diagramów klas UML i to on został wybrany do tego zadania.

4.2. Projekt interfejsu użytkownika

Projekt interfejsu użytkownika obejmował prace związane z przygotowaniem szaty graficznej systemu Wellhicle. W tej części opisano wykorzystane narzędzia, elementy identyfikacji wizualnej oraz wybrane widoki opracowane na potrzeby projektu.

4.2.1. Figma

Figma jest opartym na chmurze, działającym w przeglądarce narzędziem do projektowania interfejsów i doświadczeń użytkownika, służącym do tworzenia stron internetowych, aplikacji mobilnych oraz prototypów aplikacji [16] [17] .

Została wykorzystana do opracowania projektu interfejsu użytkownika oraz przygotowania prototypów wybranych widoków systemu. Pozwoliło to na wcześniejsze zdefiniowanie układu ekranów, sposobu prezentacji informacji oraz zachowania podstawowych komponentów interfejsu. Zastosowanie tego narzędzia wspierało zachowanie spójności projektowej, a także ułatwiało późniejsze odwzorowanie przyjętych rozwiązań w implementacji aplikacji.

4.2.2. Inkscape

Inkscape to darmowy program o otwartym kodzie źródłowym, przeznaczony do tworzenia i edycji grafiki wektorowej [18].

Narzędzie to wykorzystano do opracowania kilku wariantów logo aplikacji. Zastosowanie grafiki wektorowej ułatwiło późniejsze wykorzystanie znaku w różnych częściach systemu bez utraty jakości.

4.2.3. Material Design 3

Material Design 3 to otwarty system projektowy firmy Google, obejmujący wytyczne projektowe, komponenty, style wizualne, kolorystykę oraz typografię [19].

Został wybrany jako podstawa projektu interfejsu, ponieważ ułatwiał przygotowanie spójnych widoków bez konieczności definiowania wszystkich reguł wizualnych od podstaw. Znaczenie miała także możliwość zastosowania tego samego podejścia w interfejsie przeglądarkowym i mobilnym oraz dostępność oficjalnych materiałów projektowych dla Figmy.

4.2.4. Material Theme Builder

Material Theme Builder to wtyczka do Figmy przeznaczona do konfiguracji palety kolorów oraz typografii w systemie projektowym Material Design 3 [20].

Została wykorzystana przy dostosowywaniu gotowego zestawu zasad Material Design 3 do potrzeb systemu Wellhicle. Za pomocą tego narzędzia edytowano przede wszystkim kolorystykę oraz typografię interfejsu. Pozwoliło to zachować strukturę i spójność gotowego design systemu, a jednocześnie dopasować jego wygląd do identyfikacji wizualnej projektowanej aplikacji.

4.3. Serwer aplikacyjny

Ten podrozdział przedstawia narzędzia, biblioteki i platformy tworzące stos technologiczny serwera aplikacyjnego Wellhicle.

4.3.1. Visual Studio i JetBrains Rider

Visual Studio i JetBrains Rider to zintegrowane środowiska programistyczne (ang. *Integrated Development Environment – IDE*) przeznaczone do tworzenia aplikacji na platformie .NET [21] [22]. Oba zapewniają pełne wsparcie dla języka C#, debugowanie, refaktoryzację, analizę kodu oraz pracę z rozwiązaniami wieloprojektowymi. Działają jako warstwa robocza programisty, integrując edycję kodu, budowanie projektu, uruchamianie aplikacji i diagnostykę w jednym środowisku. W zespole zostały użyte równolegle Visual Studio na systemie Windows i Rider jako alternatywa wieloplatformowa.

4.3.2. Git

Git jest rozproszonym systemem kontroli wersji służącym do zarządzania zmianami w kodzie źródłowym [23]. Umożliwia bezpieczne rozwijanie projektu, śledzenie historii zmian i równoległą pracę nad nowymi funkcjonalnościami, dzięki czemu ułatwia pracę zespołową. Działa przez zapisywanie kolejnych wersji stanu repozytorium oraz scalanie zmian pochodzących z różnych gałęzi rozwoju.

4.3.3. SQL Server Management Studio

SQL Server Management Studio (SSMS) to narzędzie administracyjne przeznaczone do pracy z bazą danych Microsoft SQL Server [24]. Ułatwia analizę schematu, bezpośrednie wykonywanie zapytań SQL i weryfikację danych podczas rozwoju aplikacji. Działa jako graficzna nakładka administracyjna pozwalająca komunikować się z serwerem bazy danych i wykonywać operacje diagnostyczne.

4.3.4. Docker i Docker Compose

Docker to platforma do uruchamiania aplikacji w izolowanych kontenerach [25]. Docker Compose to narzędzie do definiowania i uruchamiania wielu powiązanych usług jednocześnie [26]. Razem upraszczają przygotowanie lokalnego środowiska uruchomieniowego i pozwalają zachować spójność konfiguracji infrastruktury pomiędzy stanowiskami roboczymi członków zespołu. Działają przez uruchamianie gotowych obrazów usług, takich jak SQL Server i RabbitMQ, na podstawie zadeklarowanej konfiguracji. Eliminuje to problem ręcznej instalacji wszystkich zależności w systemie operacyjnym

4.3.5. C#

C# jest nowoczesnym, statycznie typowanym językiem programowania rozwijanym w ekosystemie platformy .NET [27]. Wybór wynikał z dobrego dopasowania języka do implementacji aplikacji biznesowych. Zapewnia czytelność kodu, bezpieczeństwo typów i szerokie wsparcie

narzędziowe. Kompiluje się do kodu pośredniego wykonywanego następnie przez środowisko uruchomieniowe .NET.

4.3.6. .NET 8

.NET 8 jest platformą uruchomieniową i programistyczną wykorzystywaną do budowy aplikacji serwerowych, desktopowych i usługowych [28]. Charakteryzuje się wysoką wydajnością, dojrzałością ekosystemu oraz statusem wersji o długoterminowym wsparciu (ang. *Long-Term Support*). Stanowi środowisko wykonawcze i zestaw bibliotek bazowych, na których opierają się wszystkie główne moduły aplikacji serwerowej.

4.3.7. Microsoft SQL Server

Microsoft SQL Server to relacyjny system zarządzania bazą danych [29]. Dobrze współpracuje z platformą .NET, zapewnia obsługę transakcji, relacji, indeksów oraz ograniczeń integralności. Jest to niezbędne w systemie operującym na silnie powiązanych danych biznesowych. Stanowi centralny magazyn danych dla użytkowników, tenantów, pojazdów, rezerwacji, płatności, dokumentów oraz pozostałych jednostek biznesowych systemu. Rozważano PostgreSQL, jednak SQL Server lepiej wpisywał się w pozostały ekosystem technologiczny zastosowany w projekcie.

4.3.8. Entity Framework Core

Entity Framework Core jest obiektowo-relacyjnym maperem (ang. *Object-Relational Mapper – ORM*) pośredniczącym pomiędzy kodem aplikacji a relacyjną bazą danych [30]. Upraszcza implementację warstwy persystencji, wspiera migracje schematu, mapowanie encji oraz wzorce repozytorium i jednostki pracy. Tłumaczy operacje wykonywane na obiektach domenowych na zapytania SQL oraz synchronizuje zmiany z bazą danych.

4.3.9. ASP.NET Core

ASP.NET Core to framework służący do budowy nowoczesnych aplikacji przeglądarkowych i interfejsów HTTP API [31]. Zapewnia wydajną obsługę żądań, mechanizm routingu, łańcuch middleware, kontrolery oraz łatwą integrację z uwierzytelnianiem, autoryzacją i dokumentacją API. Stanowi warstwę pośredniczącą pomiędzy klientem a logiką aplikacyjną, przetwarzającą każde żądanie w zdefiniowanym łańcuchu komponentów.

4.3.10. FluentValidation

FluentValidation jest biblioteką służącą do definiowania reguł walidacji danych wejściowych poza kodem kontrolerów i logiki biznesowej [32]. Wyrażenie reguł w czytelnej, deklaratywnej postaci poprawia czytelność i ułatwia utrzymanie kodu. Uruchamia zdefiniowane reguły dla obiektów żądań przed wykonaniem właściwej logiki aplikacyjnej. Jest dogodniejsza niż walidacja oparta wyłącznie na atrybutach modeli lub ręcznym sprawdzaniu danych w kontrolerach, ponieważ rozdziela zagadnienia i lepiej skaluje się na złożone reguły walidacyjne.

4.3.11. OpenAPI

OpenAPI to standard opisu interfejsów HTTP API w postaci ustrukturyzowanej dokumentacji [33]. Pozwala utrzymywać dokumentację punktów końcowych w sposób powiązany z kodem i ogranicza ryzyko rozbieżności pomiędzy implementacją a opisem interfejsu. Generuje opis operacji, modeli danych oraz mechanizmów bezpieczeństwa na podstawie konfiguracji aplikacji i kontrolerów.

4.3.12. AWSSDK.S3

AWSSDK.S3 to oficjalny klient .NET dla usług zgodnych z protokołem Amazon S3 [34]. W projekcie wykorzystywany jako klient dla samodzielnie hostowanego magazynu plików RustFS, który serwuje interfejs zgodny z S3. Pozwala używać standardowego API do operacji wgrywania, pobierania oraz zarządzania zbiorami bez konieczności pisania własnego klienta HTTP. Wybór tego klienta wynikał ze ścisłej zgodności protokołowej. Magazyn plików można w razie potrzeby wymienić na inny serwis zgodny z S3 (np. MinIO lub Amazon S3) bez modyfikacji warstwy aplikacyjnej.

4.3.13. SixLabors.ImageSharp

ImageSharp jest wieloplatformową biblioteką .NET do operacji na obrazach rastrowych [35]. W projekcie wykorzystywana do walidacji i przetwarzania plików graficznych przesyłanych przez użytkowników (zdjęcia pojazdów, dokumentacja stanu technicznego) oraz do osadzania podpisów w generowanych dokumentach. Działa w kodzie w pełni zarządzanym, bez zależności od bibliotek natywnych, co upraszcza wdrażanie aplikacji w kontenerach Linux. Alternatywą była biblioteka System.Drawing.Common, jednak ImageSharp jest oficjalnie wspierany na wszystkich platformach docelowych .NET, podczas gdy System.Drawing.Common ma ograniczenia poza systemem Windows.

4.3.14. DocumentFormat.OpenXml

DocumentFormat.OpenXml to oficjalny pakiet SDK firmy Microsoft do pracy z dokumentami w formatach Office Open XML – *DOCX*, *XLSX*, *PPTX* [36]. W projekcie używany do przetwarzania szablonów Word zawierających pola scalania (ang. merge fields), które są podstawą generowania dokumentów rozliczeniowych oraz protokołów wydania i zwrotu pojazdu. Operuje bezpośrednio na strukturze XML dokumentu, dzięki czemu nie wymaga obecności pakietu Microsoft Office na maszynie generującej plik. Alternatywą było generowanie dokumentów od zera za pomocą prostszych bibliotek, jednak praca na szablonach OOXML oddziela warstwę treści (kod aplikacji) od warstwy wizualnej (szablon utrzymywany w edytorze Word).

4.3.15. iText

iText jest biblioteką służącą do generowania i przetwarzania dokumentów PDF dla platformy .NET [37]. W projekcie wykorzystywana do operacji na plikach PDF. Działa wyłącznie w warstwie .NET, bez zależności od silnika przeglądarki. Konwersja przygotowanego szablonu DOCX do finalnego dokumentu PDF realizowana jest przez wydzieloną usługę zewnętrzną wywoływaną przez HTTP, co odciąża proces główny od kosztownego renderowania.

4.3.16. RabbitMQ

RabbitMQ jest brokerem wiadomości wykorzystywanym do komunikacji asynchronicznej pomiędzy komponentami systemu [38]. Odciąża główne API i pozwala przenieść wybrane operacje poza bezpośredni cykl żądanie-odpowiedź, co poprawia responsywność aplikacji. Umieszcza komunikaty w kolejkach, z których są one następnie odbierane i przetwarzane przez osobne komponenty w projekcie, m.in. przez workera odpowiedzialnego za wysyłkę wiadomości e-mail. Alternatywą były Azure Service Bus, Apache Kafka oraz bezpośrednie wywołania synchroniczne, jednak RabbitMQ był wystarczający i prostszy dla wymagań systemu.

4.3.17. Hangfire

Hangfire to biblioteka służąca do wykonywania zadań cyklicznych i opóźnionych w aplikacjach .NET [39]. Umożliwia harmonogramowanie procesów bez potrzeby budowania osobnego mechanizmu planowania zadań. Zapisuje informacje o zadaniach w trwałym magazynie danych (w projekcie, w bazie SQL Server) i uruchamia je zgodnie z zadanym harmonogramem lub na żądanie aplikacji. Rozważano użycie systemowych zadań cron lub zewnętrznych schedulerów, jednak Hangfire lepiej integruje się z aplikacją ASP.NET Core oraz z już używaną bazą danych.

4.3.18. IMemoryCache

IMemoryCache to wbudowany w ASP.NET Core mechanizm pamięci podręcznej działającej w pamięci procesu aplikacji [40]. W modelu pojedynczej instancji serwera zapewnia prosty i wystarczający sposób przyspieszenia często wykonywanych operacji pomocniczych, m.in. rozpoznawania tenantów na podstawie subdomeny oraz obsługi danych związanych z bezpieczeństwem tokenów. Przechowuje obiekty w pamięci aplikacji przez określony czas i zwraca je bez konieczności ponownego odwołania do bazy danych. W przypadku przeniesienia aplikacji na wiele instancji wykorzystanie IMemoryCache należałoby zastąpić rozproszonym magazynem działającym w pamięci operacyjnej, takim jak Redis.

4.3.19. JWT

JWT (ang. *JSON Web Token*) to standard tokenów służących do przenoszenia informacji o tożsamości i uprawnieniach użytkownika [41]. Pozwala utrzymać warstwę serwerową w modelu bezstanowym, upraszcza obsługę sesji i dobrze pasuje do architektury HTTP API. Klient dołącza podpisany token do każdego żądania, a serwer weryfikuje jego poprawność i odczytuje zawarte w nim dane. Alternatywą była klasyczna sesja serwerowa oraz delegacja do zewnętrznego dostawcy tożsamości, jednak własna implementacja oparta na JWT była lepiej dopasowana do wymagań projektu i kontroli nad mechanizmem unieważniania tokenów.

4.3.20. Microsoft.AspNetCore.Identity PasswordHasher

PasswordHasher to komponent z pakietu *Microsoft.AspNetCore.Identity* służący do bezpiecznego hashowania i weryfikacji haseł użytkowników [42]. Eliminuje konieczność samodzielnej implementacji ochrony haseł i opiera bezpieczeństwo na sprawdzonym mechanizmie z ekosystemu ASP.NET. Tworzy skrót kryptograficzny hasła wraz z parametrami algorytmu, a następnie porównuje go z wartością zapisaną w bazie podczas weryfikacji.

4.3.21. Serilog

Serilog jest biblioteką logowania strukturalnego dla aplikacji .NET [43]. Pozwala wzbogacać logi o pola kontekstowe, m.in. identyfikator tenanta i identyfikator korelacji żądania. Ułatwia to analizę błędów, filtrowanie zdarzeń i korelację informacji pochodzących z różnych części systemu. Zapisuje logi jako ustrukturyzowane rekordy kierowane do wybranych źródeł, takich jak konsola, plik czy system agregacji logów. Alternatywą był domyślny mechanizm logowania platformy oraz NLog, jednak Serilog okazał się dogodniejszy przy logowaniu wielotenantowym i diagnostyce procesów w tle.

4.3.22. Microsoft.Extensions.Hosting i Dependency Injection

Microsoft.Extensions.Hosting wraz z wbudowanym kontenerem wstrzykiwania zależności (ang. *Dependency Injection*) tworzą podstawę uruchamiania i składania aplikacji .NET [44]. Umożliwiają spójną konfigurację usług, hosta aplikacji oraz zależności zarówno w głównym API, jak i w osobnym workerze pocztowym. Rejestrują usługi w kontenerze i dostarczają je do klas w momencie tworzenia obiektów przez środowisko uruchomieniowe.

4.3.23. Microsoft.Extensions.Configuration i wzorzec Options

Microsoft.Extensions.Configuration to mechanizm zarządzania konfiguracją aplikacji, a wzorzec Options pozwala wiązać dane konfiguracyjne z silnie typowanymi klasami [45]. Razem porządkują sposób odczytu ustawień środowiskowych i umożliwiają bezpieczne przekazywanie konfiguracji do poszczególnych komponentów. Odczytują sekcje z plików konfiguracyjnych i mapują je na klasy opcji wstrzykiwane następnie do usług aplikacyjnych.

4.3.24. Health Checks

Health Checks to mechanizm diagnostyczny wbudowany w ASP.NET Core, służący do sprawdzania dostępności kluczowych elementów środowiska uruchomieniowego [46]. Pozwala szybko wykrywać problemy z bazą danych, brokerem wiadomości i innymi zależnościami infrastrukturalnymi. Wykonuje zdefiniowane testy kontrolne i udostępnia ich wyniki w formie publicznego punktu diagnostycznego, który następnie jest wykorzystywany przez orkiestrator Kubernetes do oceny gotowości i kondycji instancji aplikacji.

4.3.25. Polly

Polly to biblioteka przeznaczona do implementacji polityk odpornościowych w aplikacjach .NET [47]. Wykorzystana w workerze pocztowym, który komunikuje się z infrastrukturą zewnętrzną, brokerem wiadomości oraz serwerami SMTP, i musi być odporny na błędy przejściowe. Opakowuje operacje w polityki takie jak ponawianie prób, wyłącznik (ang. *circuit breaker*) oraz ograniczenie czasu wykonania. Alternatywą była ręczna implementacja mechanizmów retry i circuit breaker lub użycie rozbudowanych frameworków integracyjnych, jednak Polly oferuje prosty i wystarczający model dla potrzeb projektu.

4.3.26. libphonenumber-csharp

libphonenumber-csharp [48] to portowana na .NET wersja biblioteki Google libphonenumber [49], służąca do walidacji i normalizacji numerów telefonów. Pozwala korzystać ze sprawdzonych, regularnie aktualizowanych reguł formatowania numerów dla poszczególnych krajów zamiast implementować je samodzielnie. Analizuje numer telefonu zgodnie z regułami przypisanymi do konkretnego kraju i zwraca jego znormalizowaną reprezentację. Alternatywą była walidacja oparta na wyrażeniach regularnych, jednak takie podejście byłoby mniej kompletne i trudniejsze w utrzymaniu.

4.4. Interfejs przeglądarkowy

W tym punkcie opisano podstawowe narzędzia wykorzystane do implementacji aplikacji przeglądarkowych wraz z uzasadnieniem ich wyboru. Narzędziom specyficznym dla poszczególnych widoków poświęcono osobne podpunkty w podrozdziale opisującym implementację aplikacji przeglądarkowych.

4.4.1. Angular 20

Angular to *framework* dostarczający zestaw narzędzi, bibliotek oraz API, które upraszczają proces tworzenia skalowalnych aplikacji internetowych [50]. Jego architektura oparta jest na komponentach oraz mechanizmie wstrzykiwania zależności, co umożliwia tworzenie modularnego i łatwego w utrzymaniu kodu. Trzy najczęściej używane *frameworki frontendowe* według rankingu Stack Overflow to: React, Angular i Vue.js [51]. Spośród nich wybrano Angulara, ponieważ jako jedyny dostarcza po stronie klienta kompletne rozwiązanie z wbudowanym routingiem, obsługą formularzy i klientem HTTP.

4.4.2. TypeScript

TypeScript jest nadzbiorem języka JavaScript, wprowadzającym opcjonalne statyczne typowanie. Zastosowanie typów umożliwia wykrywanie błędów na etapie kompilacji, co zwiększa niezawodność aplikacji [52]. Wykorzystanie w projekcie TypeScriptu wynikało bezpośrednio z zastosowania Angulara, który jest z nim ściśle zintegrowany i wykorzystuje go jako podstawowy język tworzenia aplikacji.

4.4.3. RxJS

RxJS to biblioteka do programowania reaktywnego opartego na strumieniach danych. Pozwala obsługiwać operacje asynchroniczne i zdarzenia za pomocą strumieni, które emitują wartości w czasie [53]. Począwszy od wersji 16, Angular wprowadził sygnały jako nowy mechanizm reaktywny, co ograniczyło rolę RxJS głównie do obsługi asynchronicznych strumieni danych. Wbudowane API klienta HTTP nadal domyślnie zwraca obiekty typu Observable, dlatego w projekcie RxJS zastosowano wszędzie tam, gdzie wynikało to z interfejsu *frameworka*.

4.4.4. Angular Material

Angular Material to biblioteka komponentów interfejsu użytkownika rozwijana przez ten sam zespół co Angular, zaprojektowana z myślą o bezpośredniej integracji z Angularem [54]. System

motywów biblioteki oparty jest na tokenach. Zdefiniowanie palety kolorów, typografii i gęstości komponentów przekłada się na zestaw zmiennych CSS, które kontrolują wygląd całej aplikacji [55]. Pozwala to na spójną i scentralizowaną konfigurację stylów bez konieczności nadpisywania wewnętrznych *selectorów*. Biblioteka implementuje specyfikację Material Design 3 (MD3), dla której Google udostępnia oficjalny zestaw komponentów w Figmie. Umożliwiło to tworzenie w ramach projektu makiet interfejsu w oparciu o te same komponenty i tokeny kolorystyczne, które następnie zastosowano w kodzie aplikacji. Wspólny wzorzec nie objął jednak komponentów Angular Material wykraczających poza specyfikację Material Design 3.

4.4.5. ESLint

ESLint to powszechnie używane narzędzie do statycznej analizy kodu, które wskazuje potencjalne błędy oraz pilnuje przestrzegania przyjętych reguł [56]. Zastosowano je w celu utrzymania spójności, czytelności kodu oraz stosowania dobrych praktyk programistycznych. W konfiguracji wykorzystano zestawy reguł zalecanych dla języka TypeScript oraz dla Angulara, pochodzące z biblioteki angular-eslint. Reguły uzupełniono o konwencje nazewnictwa *selectorów*, komponentów i dyrektyw obowiązujące w projekcie. Analizę uruchamiano automatycznie w procesie CI/CD, gdzie wykryte naruszenia wstrzymywały scalenie zmian.

4.4.6. Prettier

Prettier to formater, który ujednolica styl kodu według ustalonego zestawu zasad [57]. Wybrano go ze względu na kompatybilność z ESLintem, dzięki której naruszenia formatowania zgłaszane są jako błędy analizy. Pozwala to weryfikować spójność stylu i poprawność kodu w ramach jednego procesu. Dodatkowo jednolite formatowanie ułatwia przeglądanie zmian w systemie kontroli wersji Git.

4.4.7. Ngx-translate

Ngx-translate to biblioteka umożliwiająca dynamiczną zmianę języka w czasie wykonania, bez konieczności przeładowania aplikacji [58]. Alternatywę stanowił wbudowany w Angulara mechanizm @angular/localize, który wymaga kompilacji aplikacji osobno dla każdego z obsługiwanych języków [59]. Z tego względu nie spełniał wymagań projektu. W plikach tłumaczeń przyjęto konwencję porządkującą wpisy według ich zasięgu. Etykiety używane w wielu miejscach aplikacji zapisano jako klucze na najwyższym poziomie. Teksty właściwe dla danego komponentu zgrupowano pod kluczem jego nazwy. Osobno zgrupowano wartości enumów.

4.5. Interfejs mobilny

W poniższym podrozdziale zostaną opisane podstawowe narzędzia i biblioteki wykorzystane do implementacji aplikacji mobilnej w ramach systemu Wellhicle. Rozwiązaniom, które wymagają szerszego omówienia zostaną poświęcone osobne podrozdziały.

4.5.1. Kotlin Multiplatform

Kotlin Multiplatform (KMP) to technologia umożliwiająca współdzielenie kodu aplikacji między różnymi platformami – Android , iOS, Desktop i Web – przy jednoczesnym zachowaniu dostępu do natywnych interfejsów programistycznych każdej z nich [60]. Logika biznesowa, warstwa danych i

warstwa sieciowa wyodrębnione są do wspólnego modułu commonMain , kompilowanego niezależnie do kodu bajtowego JVM dla Androida oraz do natywnego kodu maszynowego dla iOS. Fragmenty wymagające platformo-specyficznych API deklaruje się za pomocą mechanizmu expect/actual: deklaracja wspólna w commonMain jest uzupełniana o konkretne implementacje w katalogach androidMain i iosMain . Aplikacja Wellhicle zbudowana jest przy użyciu Kotlin 2.2.20 i przeznaczona na platformy Android (minimalna wersja API 24, docelowa API 36) oraz iOS.

4.5.2. Compose Multiplatform

Warstwa prezentacji zrealizowana jest przy użyciu Compose Multiplatform 1.9.1. Jest to *framework* UI rozwijający Jetpack Compose na środowisko wieloplatformowe, umożliwiający utrzymanie jednego wspólnego kodu interfejsu użytkownika dla platform Android i iOS z pojedynczego źródła kodu [61]. Renderowanie odbywa się natywnie na każdej z platform [62], co odróżnia to podejście od rozwiązań opartych na WebView. Alternatywę stanowił model natywny, w którym widoki dla Androida byłyby pisane w Jetpack Compose, a dla iOS w SwiftUI – takie podejście wymagałoby jednak powielenia całej warstwy UI i synchronizowania dwóch niezależnych implementacji. Compose Multiplatform pozwala pisać ekrany raz, w module commonMain , przy czym renderowanie odbywa się natywnie na każdej platformie.

4.5.3. Ktor

Do komunikacji sieciowej z serwerem REST API systemu Wellhicle zastosowano Ktor 3.3.1 jako klienta HTTP. Ktor to asynchroniczna biblioteka klienta HTTP zaprojektowana dla Kotlin Multiplatform [63]. Biblioteka dostarcza dedykowane silniki sieciowe dla każdej obsługiwanej platformy: ktor-client-okhttp oparty o bibliotekę OkHttp dla Androida oraz ktor-client-darwin korzystający z natywnego mechanizmu NSURLSession dla iOS [64]. Alternatywą był Retrofit – jednak jest to biblioteka JVM/Android-only i nie jest kompatybilna z Kotlin Multiplatform; nie posiada implementacji dla iOS. Ktor dostarcza oficjalny silnik dla każdej platformy, dzięki czemu ten sam kod definiujący żądania HTTP działa bez modyfikacji w commonMain, a wybór silnika sieciowego następuje w czasie kompilacji. Do serializacji danych zastosowano kotlinx-serialization v1.9.0, które w odróżnieniu od Gson i Moshi, jest w pełni kompatybilne z Kotlin Multiplatform i obsługuje kompilację do natywnego kodu iOS [65]. Asynchroniczność i obsługa operacji w tle realizowana jest przez kotlinx-coroutines v1.10.2.

4.5.4. Koin

Zarządzanie zależnościami oparto o *Koin* 4.1.0. *Koin* to pragmatyczna, lekka biblioteka wstrzykiwania zależności dla *Kotlin* i *Kotlin Multiplatform*, niewymagająca przetwarzania adnotacji w czasie kompilacji [66]. Wstrzykiwanie odbywa się w czasie wykonania przez rejestry modułów zdefiniowane programowo. Głównym rywalem był *Hilt* (Google), jednak *Hilt* jest biblioteką tylko dla systemu Android, a jego mechanizm przetwarzania adnotacji (*kapt*) nie jest wspierany w środowisku KMP/iOS [67].

4.5.5. Przechowywanie danych lokalnych - multiplatform-settings

Trwałe przechowywanie danych lokalnych (tokeny sesji JWT, preferencje językowe) zrealizowano przy użyciu biblioteki multiplatform-settings 1.3.0. Biblioteka dostarcza klucz – wartościowy interfejs do przechowywania danych w środowiskach *Kotlin Multiplatform*, korzystając z

mechanizmów natywnych dla każdej platformy: *SharedPreferences* na Androidzie i *NSUserDefaults* na iOS [68]. Implementacja tego narzędzia nie wymaga żadnego kodu specyficznego dla platformy iOS/Android

4.5.6. BuildKonfig

Konfiguracja środowisk aplikacji realizowana jest przez wtyczkę *BuildKonfig* v0.17.1. Wtyczka generuje typowane stałe Kotlin na podstawie parametrów *build flavor* zdefiniowanych w *Gradle*, analogicznie do mechanizmu *BuildConfig* dostępnego natywnie dla projektów Android [69]. Podejście to pozwala oddzielić adresy środowisk deweloperskiego i produkcyjnego bez konieczności ręcznego zarządzania stałymi w kodzie. Aplikacja posiada dwa warianty kompilacji: *dev* i *prod*.

4.5.7. Firebase App Distribution

Dystrybucja wersji testowych aplikacji odbywa się za pomocą *Firebase App Distribution* v5.2.0. Narzędzie umożliwia udostępnianie kompilacji aplikacji Android i iOS wskazanym testerom bez konieczności publikacji w sklepach aplikacji [70], co skraca czas dostarczania kompilacji testowych podczas iteracyjnego wytwarzania oprogramowania.

4.5.8. Navigation Compose

Nawigacja między ekranami zrealizowana jest przy użyciu biblioteki *navigation-compose 2.9.1* z grupy *org.jetbrains.androidx.navigation*. Navigation Compose to biblioteka do deklaratywnego zarządzania nawigacją w aplikacjach opartych na Jetpack Compose, udostępniająca komponent NavHost, kontroler NavHostController oraz API do definiowania grafów nawigacyjnych w Kotlinie [71]. Biblioteka pochodzi z projektu Jetpack i jest oficjalnie wspierana przez Google jako standardowe rozwiązanie nawigacyjne dla Compose [72]. Wariant *org.jetbrains.androidx.navigation* jest wieloplatformową adaptacją utrzymywaną przez JetBrains, umożliwiającą korzystanie z tego samego API nawigacyjnego zarówno na Androidzie, jak i na iOS w środowisku Compose Multiplatform. Alternatywą był Decompose – biblioteka nawigacji stworzona z myślą o KMP – jednak Navigation Compose oferuje mniejszy narzut koncepcyjny dla programistów zaznajomionych z ekosystemem Jetpack oraz pełną integrację z *ViewModel* i *lifecycle-viewmodel-compose*. Szczegółowa implementacja nawigacji opisana jest w punkcie *7.3.1*.

4.6. Wdrażanie i monitorowanie

W tym podrozdziale przedstawiono narzędzia użyte do uruchomienia, wdrażania oraz obserwowania działania systemu Wellhicle.

4.6.1. OpenTelemetry

OpenTelemetry to standard oraz zestaw bibliotek służących do gromadzenia telemetrii aplikacji, w tym śladów wykonania i metryk. Kluczowym czynnikiem wyboru był fakt, że umożliwia obserwację przepływu żądań przez system oraz diagnozowanie problemów wydajnościowych i błędów w sposób bardziej uporządkowany niż same logi. Działa poprzez instrumentację warstw aplikacji i eksportowanie danych telemetrycznych do skonfigurowanych odbiorców. Istniała możliwość ograniczenia się wyłącznie do logów lub użycia zamkniętych narzędzi monitorujących, jednak

OpenTelemetry zapewnia większą elastyczność i lepiej wpisuje się w nowoczesny model obserwowalności. Jest on standardem rynkowym, jeżeli chodzi o technologie do gromadzenia telemetrii aplikacji. [73]

4.6.2. NGINX

NGINX pełni w systemie rolę serwera HTTP, działając jako serwer WWW oraz odwrotny serwer proxy (ang. reverse proxy server). Wybór padł na niego ze względu na wydajną obsługę dużej liczby jednoczesnych połączeń przy niskim zużyciu zasobów, a także bogate możliwości konfiguracji routingu opartego na domenie. Mechanizm działania polega na przyjmowaniu przychodzących żądań HTTP i przekierowywaniu ich do odpowiednich komponentów wewnętrznych zgodnie z ustalonymi regułami. Rozważano również Traefika lub HAProxy, jednak NGINX spełniał wymagania zespołu. [74]

4.6.3. Kubernetes

Kubernetes jest systemem orkiestracji kontenerów służącym do automatycznego wdrażania, skalowania i zarządzania konteneryzowanymi aplikacjami. Są one spakowane wraz ze wszystkimi swoimi zależnościami i konfiguracją w izolowaną jednostkę uruchomieniową działającą w sposób identyczny niezależnie od środowiska. Został wybrany, ponieważ umożliwia niezależne wdrażanie poszczególnych komponentów systemu, automatyczne restarty przy awarii i deklaratywne zarządzanie stanem infrastruktury. Działa przez utrzymanie pożądanego stanu klastra opisanego w plikach manifestów i wykrywanie oraz korygowanie odchyleń od tego stanu. Alternatywą było użycie zarządzanej usługi chmurowej takiej jak Elastic Container Service na AWS lub prostego Docker Compose w środowisku produkcyjnym, jednak Kubernetes daje większą kontrolę nad środowiskiem i był znany osobie odpowiedzialnej za infrastrukturę. [75]

4.6.4. Cloudflare Tunnel

Cloudflare Tunnel jest narzędziem eliminującym konieczność bezpośredniej ekspozycji klastra Kubernetes w publicznej sieci. Został wybrany, ponieważ pozwala ukryć klaster za siecią Cloudflare bez otwierania portów wejściowych w zaporze sieciowej i bez zarządzania certyfikatami TLS po stronie klastra. Działa przez nawiązanie połączenia wychodzącego z infrastruktury klastra do sieci Cloudflare, przez które następnie kierowany jest cały ruch przychodzący. Alternatywą było skonfigurowanie tradycyjnego load balancera z publicznym adresem IP i certyfikatem TLS, jednak Cloudflare Tunnel upraszcza konfigurację sieci i przenosi terminację TLS do infrastruktury Cloudflare. [76]

4.6.5. Harbor

Harbor jest prywatnym rejestrem obrazów kontenerowych. Został wybrany, ponieważ przechowywanie obrazów kontenerowych na własnej infrastrukturze eliminuje zależność od zewnętrznych usług rejestrów przy wdrożeniu i daje pełną kontrolę nad przechowywanymi wersjami obrazów. Działa przez przyjmowanie, przechowywanie i udostępnianie obrazów kontenerowych przy użyciu protokołu Docker Registry. Alternatywą były usługi takie jak Docker Hub lub GitHub Container Registry, jednak Harbor był najsensowniejszym wyborem, jeżeli chodzi o koszt wdrożenia rozwiązania, ponieważ jest dostępny bezpłatnie. [77]

4.6.6. RustFS

RustFS jest serwerem plików zgodnym z protokołem S3, czyli interfejsem API opracowanym przez Amazon, służący do przechowywania i pobierania plików za pomocą żądań HTTP. Zastosowanie wyodrębnionego magazynu plików wynikało z charakteru przechowywanych danych. Zdjęcia pojazdów i dokumenty PDF to pliki binarne o zmiennym rozmiarze, których przechowywanie w relacyjnej bazie danych obniża wydajność zapytań, zwiększa rozmiar kopii zapasowych i utrudnia niezależne skalowanie warstwy danych. SQL Server udostępnia mechanizm FILESTREAM pozwalający przechowywać dane binarne poza tabelami, jednak nadal wiąże ich cykl życia z transakcjami bazodanowymi i wymaga dostępu przez interfejs SQL, co wprowadza zbędne powiązanie między logiką aplikacyjną a systemem zarządzania bazą danych. RustFS przyjmuje operacje na plikach za pomocą protokołu S3 i przechowuje je w skonfigurowanym wolumenie, co pozwala używać standardowego klienta AWSSDK.S3 bez dedykowanych bibliotek. Alternatywę stanowiły MinIO oraz usługa Amazon S3, jednak RustFS oferuje kompatybilne API przy niższych wymaganiach zasobowych i jest dostępny bezpłatnie przy uruchomieniu na własnej infrastrukturze. [78]

4.6.7. Prometheus

Prometheus jest systemem monitorowania i alertowania przeznaczonym do zbierania i przechowywania metryk szeregów czasowych. Uznano go za najlepsze rozwiązanie z uwagi na to, że dostarcza sprawdzony model zbierania metryk przez aktywne odpytywanie komponentów (ang. scraping) i definiowania reguł alertów. Działa przez cykliczne odpytywanie wystawionych przez komponenty punktów końcowych metryk i zapisywanie ich wartości wraz ze znacznikiem czasu. Alternatywą były systemy takie jak InfluxDB lub Datadog, jednak Prometheus jest szeroko stosowany w ekosystemie Kubernetes i dobrze integruje się z Grafaną. [79]

4.6.8. Loki

Loki jest systemem agregacji logów zoptymalizowanym pod kątem integracji z Grafaną. Został wybrany, ponieważ w odróżnieniu od systemów takich jak Elasticsearch nie indeksuje treści logów, lecz jedynie przypisane do nich etykiety metadanych, takie jak nazwa usługi czy środowisko uruchomieniowe. Dzięki temu wymaga mniej zasobów obliczeniowych i pamięci masowej, co czyni go odpowiednim rozwiązaniem dla środowiska uruchomionego na własnej infrastrukturze bez dedykowanego budżetu operacyjnego. Alternatywą było narzędzie Elasticsearch, jednak Loki lepiej integruje się z pozostałymi elementami stosu obserwowalności opartego na Grafanie. [80]

4.6.9. Tempo

Tempo jest systemem przechowywania śladów rozproszonych (ang. distributed traces) kompatybilnym z formatem OpenTelemetry. Został wybrany, ponieważ przechowuje pełne ślady wykonania operacji biznesowych i umożliwia ich przeglądanie bezpośrednio z poziomu Grafany w powiązaniu z metrykami i logami. Działa przez przyjmowanie śladów przez protokół OpenTelemetry Protocol i przechowywanie ich w sposób umożliwiający szybkie wyszukiwanie po identyfikatorze śladu. Alternatywą był Jaeger lub Zipkin, jednak Tempo lepiej integruje się z ekosystemem Grafana. [81]

4.6.10. Grafana

Grafana jest platformą do wizualizacji danych i monitorowania, łączącą dane z wielu źródeł w jednym interfejsie. Została wybrana, ponieważ umożliwia przeglądanie metryk, logów i śladów rozproszonych w jednym narzędziu i przechodzenie między nimi w kontekście tego samego zdarzenia. Działa przez połączenie z konfigurowalnymi źródłami danych, takimi jak Prometheus, Loki i Tempo, i prezentowanie ich treści w formie dashboardów i paneli diagnostycznych. Alternatywą były narzędzia takie jak Datadog lub Grafana Cloud, jednak lokalna instalacja Grafany eliminuje zależność od zewnętrznych usług i nie generuje kosztów opartych na wolumenie danych. [82]

4.6.11. K3s

K3s jest lekką dystrybucją Kubernetes przeznaczoną do uruchamiania klastrów na pojedynczym węźle lub w małych środowiskach bez pełnego stosu sterowania chmurowego. Został wybrany, ponieważ umożliwia uruchomienie środowiska produkcyjnego na własnej infrastrukturze przy mniejszym narzucie zasobowym niż standardowy klaster Kubernetes, zachowując jednocześnie kompatybilność z manifestami i narzędziami stosowanymi w ekosystemie Kubernetes. Działa przez uruchomienie uproszczonego procesu sterującego oraz wbudowanych komponentów niezbędnych do obsługi obciążeń kontenerowych, takich jak sieć i magazyn. Alternatywą było wdrożenie zarządzanego klastra w chmurze publicznej lub instalacja pełnego Kubernetes na wielu węzłach, jednak k3s okazał się wystarczający dla skali projektu i ograniczył koszty utrzymania infrastruktury. [83]

4.6.12. Helm

Helm jest menedżerem pakietów dla Kubernetes, umożliwiającym instalowanie i aktualizowanie złożonych aplikacji na podstawie gotowych wykresów (ang. *charts*) wraz z plikami wartości konfiguracyjnych. Został zaadaptowany ze względu na to, że komponenty infrastrukturalne, takie jak Harbor, stos monitorowania czy magazyn plików RustFS, są dostarczane jako wykresy utrzymywane przez społeczność, a Helm upraszcza ich wersjonowanie i powtarzalne wdrożenie w klastrze. Działa przez renderowanie szablonów manifestów Kubernetes z parametrami z pliku *values* i stosowanie wyniku w klastrze poleceniem instalacji lub aktualizacji wydania. Alternatywą było ręczne utrzymywanie pełnych manifestów YAML dla każdego komponentu, jednak Helm ogranicza duplikację konfiguracji i ułatwia aktualizacje wykresów przy zachowaniu kontroli nad kluczowymi parametrami w repozytorium projektu. [84]

4.6.13. GitHub Actions

GitHub Actions jest platformą automatyzacji zintegrowaną z repozytoriami GitHub, umożliwiającą definiowanie procesów budowania, testowania i wdrażania w plikach workflow uruchamianych na zdarzeniach w repozytorium. Został wybrany, ponieważ cały kod źródłowy projektu Wellhicle jest przechowywany na GitHubie, co pozwala powiązać proces CI/CD bezpośrednio z historią zmian, gałęziami i mechanizmem *pull request*. Działa przez wykonywanie sekwencji kroków zdefiniowanych w pliku YAML na runnerze wskazanym w konfiguracji workflow, na przykład po wypchnięciu zmian do gałęzi main lub po ręcznym wywołaniu wdrożenia. Alternatywą były zewnętrzne serwery CI, takie jak Jenkins lub GitLab CI, jednak GitHub Actions eliminuje potrzebę utrzymywania oddzielnej usługi integracyjnej i jest dobrze znany w zespole. [85]

4.6.14. Kustomize

Kustomize jest narzędziem do deklaratywnego zarządzania konfiguracją Kubernetes bez szablonów, umożliwiającym składanie manifestów z warstw bazowych i nakładek środowiskowych. Został wybrany, ponieważ repozytorium infrastructure opisuje docelowy stan klastra w jednym punkcie wejścia, łącząc surowe manifesty YAML z wykresami Helm i generowanymi zasobami, takimi jak konfiguracja serwera NGINX. Działa przez wskazanie katalogu z plikiem kustomization.yaml, który definiuje listę zasobów, generatory *ConfigMap* oraz odwołania do wykresów Helm, a następnie buduje gotowy zestaw manifestów do zastosowania w klastrze. Alternatywą było utrzymywanie wyłącznie ręcznie scalonych plików YAML lub użycie wyłącznie Helm dla całej infrastruktury, jednak Kustomize pozwala wersjonować własne manifesty projektu obok zewnętrznych wykresów i ułatwia porównywanie stanu oczekiwanego ze stanem rzeczywistym klastra. [86]

4.7. Testy

W procesie testowania systemu Wellhicle korzystano z szeregu narzędzi wspierających różne etapy i rodzaje weryfikacji – od testowania interfejsów API, przez testy automatyczne, aż po testy jednostkowe. W kolejnych punktach opisano każde z narzędzi wykorzystanych w projekcie.

4.7.1. Postman

Postman to narzędzie służące do testowania i dokumentowania interfejsów API w oparciu o protokół HTTP. Udostępnia ono graficzny interfejs, umożliwiający konstruowanie i wysyłanie zapytań, weryfikację zwracanych odpowiedzi oraz organizowanie zapytań w kolekcje [87].

W ramach projektu Wellhicle narzędzie to zostało zastosowane podczas testów manualnych i integracyjnych warstwy serwerowej systemu. Istotną funkcjonalnością wykorzystaną podczas prac był mechanizm zmiennych środowiskowych. Pozwolił on na definiowanie konfiguracji oddzielnie dla środowiska lokalnego i deweloperskiego, dzięki czemu możliwe było szybkie przełączanie się między nimi bez konieczności ręcznej modyfikacji każdego z zapytań. Zmienne przechowywały m.in. adresy bazowe oraz tokeny uwierzytelniające, co miało szczególne znaczenie przy testowaniu żądań wymagających autoryzacji.

4.7.2. Insomnia

Insomnia to narzędzie klasy klient API, które wspomaga pracę nad interfejsami aplikacji [88]. Poprzez graficzny interfejs umożliwia konstruowanie i wysyłanie zapytań HTTP oraz, podobnie jak Postman, organizowanie zapytań w kolekcje, definiowanie zmiennych środowiskowych i weryfikację odpowiedzi z serwera.

Narzędzie było wykorzystywane w początkowej fazie projektu głównie ze względu na możliwość pracy w ramach jednej organizacji, gdzie wszystkie kolekcje są synchronizowane i dostępne dla każdego z jej członków, co ułatwiało współpracę na wczesnym etapie prac. Z czasem narzędzie zaczęło sprawiać problemy. Import struktury zapytań z plików JSON oraz YAML działał niepoprawnie. Zagnieżdżona struktura żądań API była odwzorowana błędnie. Uniemożliwiało to wygodne zarządzanie zestawem zapytań. Dodatkowo lepsza znajomość Postmana przez członków zespołu wpłynęła na decyzję o zaprzestaniu wykorzystywania tego narzędzia.

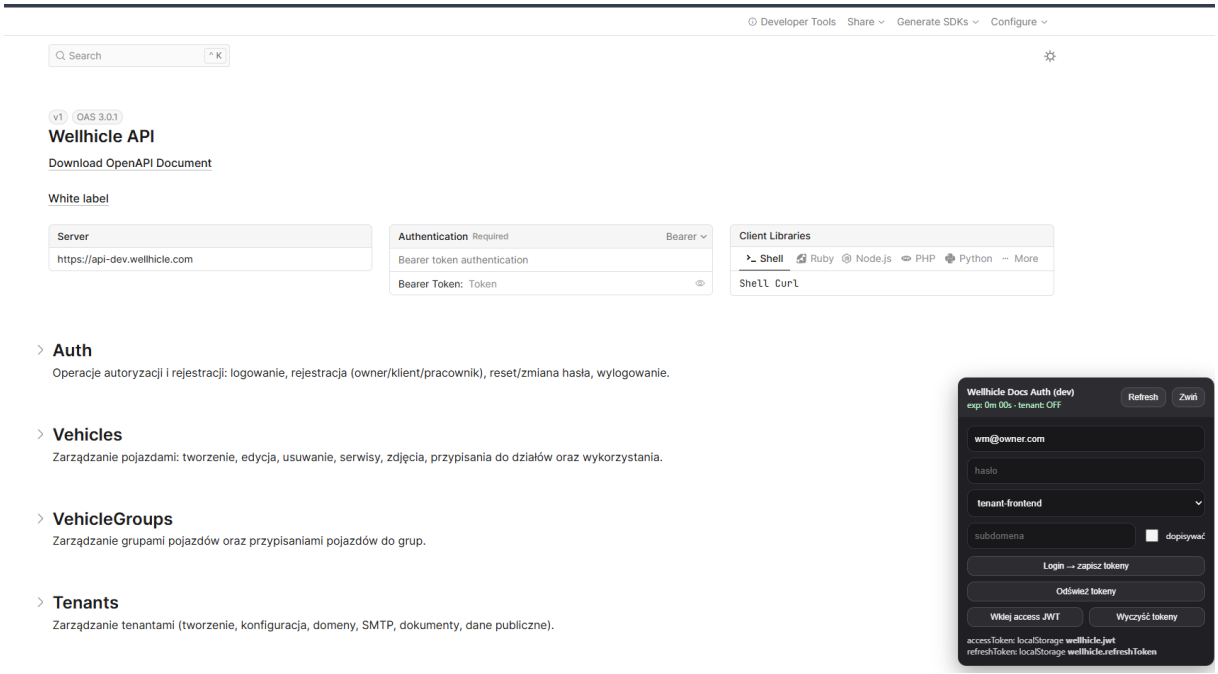
4.7.3. Swagger

Swagger to zestaw narzędzi wspomagających tworzenie dokumentacji i eksplorację API [89]. Komponent Swagger UI pozwala na wizualizację zasobów API i korzystanie z nich bezpośrednio z poziomu przeglądarki bez konieczności posiadania zewnętrznych aplikacji.

Narzędzie to było wykorzystywane w początkowej fazie projektu, natomiast wraz z postępem prac, ze względu na ograniczenia możliwości dostosowywania wyglądu oraz ilości informacji zamieszczanych w dokumentacji, zdecydowano się na bardziej zaawansowane narzędzie szerzej opisane w punkcie *4.7.4*.

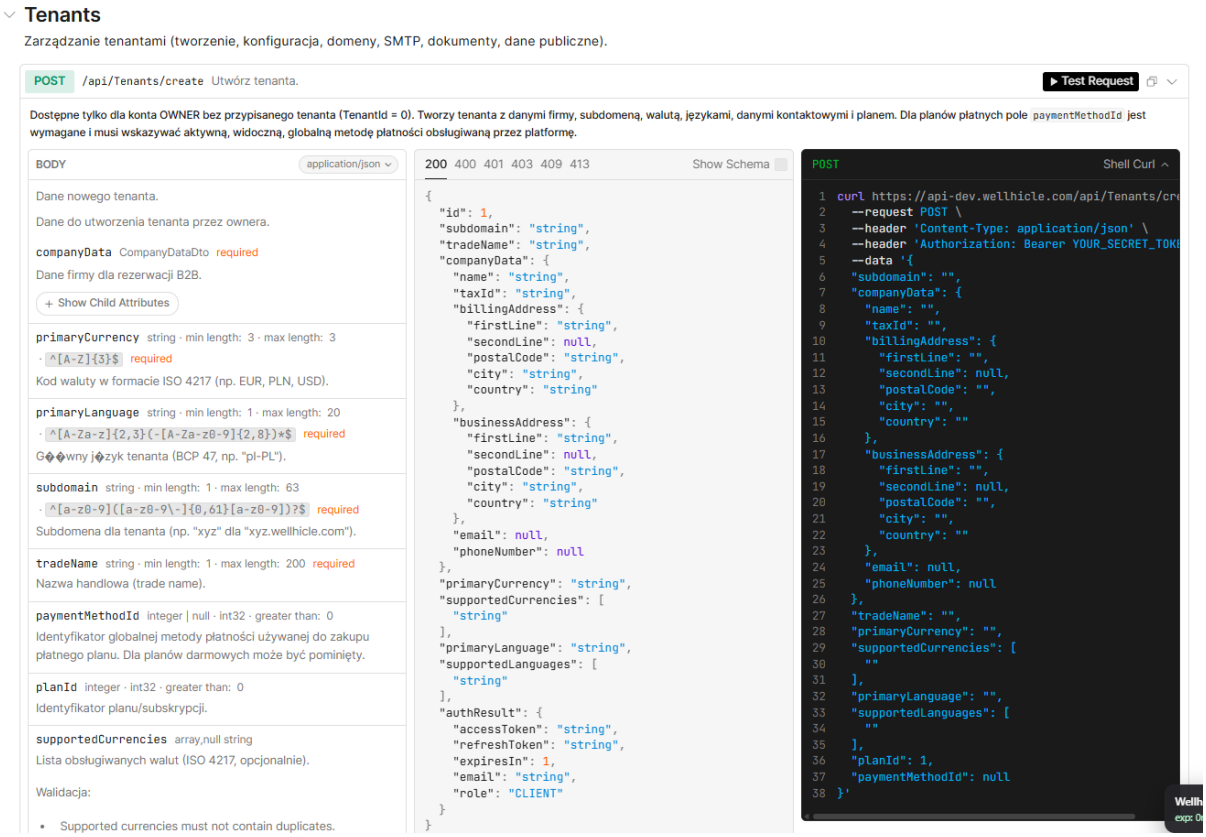
4.7.4. Scalar

Scalar to otwartoźródłowe narzędzie służące do generowania interaktywnej dokumentacji API na podstawie specyfikacji OpenAPI [90] [91]. Zapewnia czytelny interfejs, który oferuje większe możliwości w zakresie personalizacji dokumentacji oraz zamieszczanej treści w porównaniu dla alternatywy, jaką był Swagger UI. Scalar udostępnia również wbudowanego klienta HTTP, pozwalając na wysyłanie zapytań testowych bezpośrednio z poziomu dokumentacji. *Rysunek 13* przedstawia stronę powitalną Scalar z widocznymi grupami zapytań API*:* Auth, Vehicles, VehiclesGroups, Tenants.



Rysunek 13 Widok startowy Scalar. Źródło: opracowanie własne.

Rysunek 14 przedstawia widok punktu końcowego API wraz z wyszczególnieniem pól żądania i odpowiedzi, obejmującym opis zwracanego typu, ograniczeń wartości wynikających z logiki biznesowej oraz wymagalności każdego z pól.



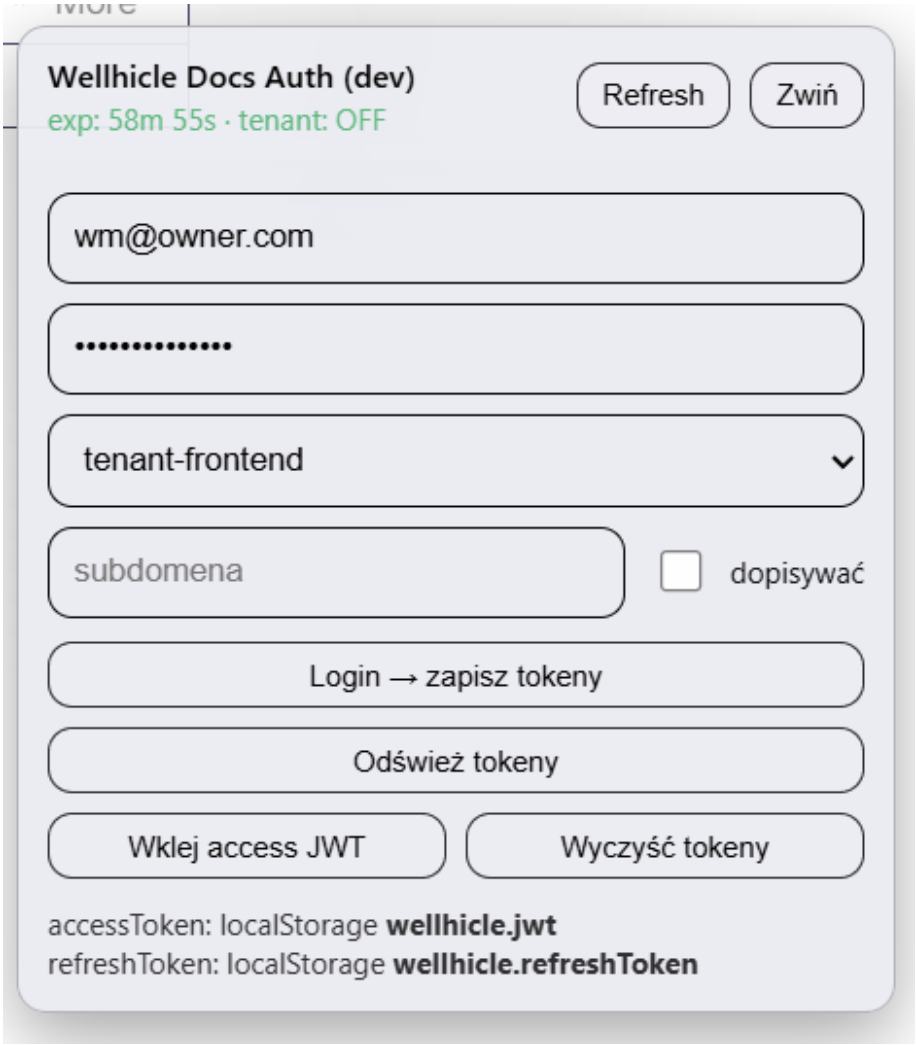
Rysunek 14 Widok pojedynczego punktu końcowego API. Źródło: opracowanie własne.

Scalar umożliwia rozszerzanie interfejsu o własne rozwiązania poprzez osadzenie dowolnego fragmentu HTML lub JavaScript na stronie dokumentacji. W toku prac nad projektem wykorzystano ten mechanizm do opracowania autorskiego rozszerzenia, którego zapotrzebowanie wyniknęło z uciążliwości towarzyszących codziennemu testowaniu API. Architektura systemu wymaga dołączania nagłówka X-Tenant-Source do większości żądań, a krótki czas życia tokenów uwierzytelniających zmuszał testerów do wielokrotnego wywoływania żądania logowania i ręcznego kopiowania tokenu między kolejnymi wywołaniami. Opracowane rozszerzenie wyeliminowało obie te uciążliwości, ponieważ automatycznie dołączało wymagane nagłówki do każdego wysyłanego żądania, a w przypadku wygaśnięcia tokenu samodzielnie wykonywało żądanie odświeżające i ponawiało pierwotne wywołanie. *Listing 1* przedstawia fragment konfiguracji osadzający skrypt w interfejsie Scalar.

Listing 1 Fragment konfiguracji osadzający autorski skrypt do narzędzia Scalar. Źródło: opracowanie własne.

```
options.HeadContent = """
    <script>
      window.__WH_DOCS__ = {
        jwtStorageKey: "wellhicle.jwt",
        loginEndpoint: "/api/auth/login"
      };
    </script>
    <script defer src="/docs-assets/wellhicle-scalar-auth.js"></script>
""";
```

Przyjęte rozwiązanie znacząco usprawniło pracę podczas testowania wieloetapowych scenariuszy biznesowych, takich jak rezerwacja pojazdu czy obsługa płatności, które wymagają wykonania wielu kolejnych żądań w obrębie jednej sesji. Dzięki rozszerzeniu testowanie API zostało zredukowane do jednorazowego wprowadzenia danych logowania. *Rysunek 15* przedstawia efekt końcowy implementacji rozszerzenia w narzędziu Scalar.



Rysunek 15 Efekt końcowy implementacji autorskiego rozwiązania w Scalar. Źródło: opracowanie własne.

4.7.5. Playwright

Playwright to otwartoźródłowe narzędzie rozwijane przez Microsoft, służące do pisania testów automatycznych interfejsu użytkownika aplikacji przeglądarkowych. Narzędzie wspiera języki TypeScript, JavaScript oraz C#, a jego działanie opiera się na koncepcji automatycznego oczekiwania, która gwarantuje wykonanie akcji testowych dopiero w momencie, gdy dany element interfejsu jest w pełni załadowany i gotowy do interakcji, redukując tym samym potrzebę ręcznego definiowania warunków oczekiwania w kodzie testów. Narzędzie umożliwia przechwytywanie i monitorowanie ruchu sieciowego podczas wykonywanych testów oraz generowanie czytelnych raportów co usprawnia proces weryfikacji błędów [92].

4.7.6. Biblioteka xUnit

XUnit to biblioteka przeznaczona do tworzenia testów jednostkowych w środowisku .NET. Posiada wbudowany mechanizm asercji pozwalający na czytelne porównywanie wartości rzeczywistych z oczekiwanymi, a komunikaty o błędach ułatwiają szybką lokalizację przyczyny niepowodzenia testu [93]. Narzędzie dostarcza dwa podstawowe atrybuty definiowania metod „Fact” dla metod testów bez parametrów oraz „Theory” dla metod parametryzowanych uruchamianych wielokrotnie dla różnych

zestawów danych wejściowych. W projekcie Wellhicle biblioteka stanowiła podstawę całego pokrycia testowego warstwy serwerowej.

5. Analiza

Analiza wyznacza obraz tego, co system Wellhicle ma realizować i dla kogo. Rozdział opisuje warstwę biznesową systemu: jego odbiorców, oczekiwane funkcje oraz reguły obowiązujące w dziedzinie wypożyczania pojazdów, a także porządkuje pojęcia, którymi posługują się kolejne rozdziały pracy. Tak ustalony, jednolity obraz dziedziny stanowi podstawę dla dalszych etapów realizacji.

Rozdział pozostaje przy tym na poziomie biznesowym - opisuje oczekiwane zachowanie systemu, nie rozstrzygając jego architektury ani sposobu realizacji wymagań; te zagadnienia należą do dalszych rozdziałów pracy.

Wykaz skrótów artefaktów i modułów

Każdy artefakt analizy ma identyfikator w postaci *<TYP>-<XX>-<NN>*, gdzie *<TYP>* oznacza rodzaj artefaktu, *<XX>* to dwu- lub trzyznakowy klucz określający przynależność artefaktu (skrót modułu funkcjonalnego dla większości typów albo kod kategorii dla wymagań niefunkcjonalnych), a *<NN>* to dwucyfrowy numer porządkowy w obrębie tego klucza. Liczniki są osobne dla każdego typu artefaktu – *WF-REZ-01* i *RB-REZ-01* to dwa różne artefakty. Rodzaje artefaktów stosowane w rozdziale zebrane są w *Tabela 10*, a skróty modułów funkcjonalnych – w *Tabela 11*.

Tabela 10 Rodzaje artefaktów analizy i ich oznaczenia. Źródło: opracowanie własne.

Skrót	Artefakt	Sekcja
WF	Wymaganie funkcjonalne	*5.3*
RB	Reguła biznesowa i walidacja	5.3
NF	Wymaganie niefunkcjonalne	*5.4*
UC	Przypadek użycia	5.5
AD	Diagram aktywności	*5.6*
CD	Analityczny model dziedziny (klasy)	*5.7*

Artefakty typu *UC*, *WF*, *RB*, *AD* i *CD* w polu *<XX>* używają trzyliterowego skrótu modułu funkcjonalnego z *Tabela 11*. Wymagania niefunkcjonalne (*NF*) - z uwagi na przekrojowy charakter, niezwiązany z pojedynczym modułem - w polu *<XX>* używają dwuliterowego kodu kategorii (wydajność, bezpieczeństwo, skalowalność i inne), opisanego w podrozdziale *5.4*.

Tabela 11 Moduły funkcjonalne systemu i ich skróty. Źródło: opracowanie własne.

Nr	Moduł	Skrót
5.3.1	Zarządzanie wypożyczalnią	ZWY
5.3.2	Flota	FLO
5.3.3	Klienci	KLI
5.3.4	Oferty	OFE
5.3.5	Rezerwacje	REZ
5.3.6	Finanse	FIN
5.3.7	Płatności	PLA
5.3.8	Operacyjny	OPE
5.3.9	Strona publiczna wypożyczalni	SWY
5.3.10	Pakiety	PAK
5.3.11	Administracja platformy	ADM
5.3.12	Komunikacja	KOM
5.3.13	Strona publiczna platformy	SPL

5.1. Streszczenie

Przedmiotem analizy jest platforma udostępniana w modelu SaaS (ang. *Software as a Service*), służąca do kompleksowego prowadzenia wypożyczalni pojazdów. Istotą proponowanego rozwiązania jest oddzielenie dwóch perspektyw, które w samodzielnych aplikacjach bywają mylone. Z jednej strony platforma obsługuje wypożyczalnie jako swoich klientów platformy - to one wykupują dostęp do systemu Wellhicle, aby za jego pomocą prowadzić własną działalność. Z drugiej strony każda wypożyczalnia obsługuje za pośrednictwem platformy swoich klientów wypożyczalni - osoby rezerwujące i wynajmujące pojazdy. Rozróżnienie między klientem platformy a klientem wypożyczalni jest kluczowe dla zrozumienia całego rozdziału i powraca w katalogu aktorów, wymaganiach oraz regułach biznesowych.

Platforma obsługuje wiele wypożyczalni jednocześnie, a każda z nich prowadzi działalność niezależnie od pozostałych - dysponuje Został wybrany, ponieważ, flotą, ofertą oraz bazą klientów wypożyczalni i udostępnia swoją stronę pod własnym adresem internetowym. Obsługa wielu niezależnych wypożyczalni w ramach jednej platformy jest nadrzędnym założeniem produktu, do którego nawiązują dalsze sekcje rozdziału.

Zakres systemu jest szeroki i obejmuje dwie warstwy. W warstwie operacyjnej – z perspektywy pojedynczej wypożyczalni – system pokrywa pełny cykl jej działalności: umożliwia skonfigurowanie organizacji wraz z oddziałami, walutą i obsługiwanymi językami, prowadzenie floty pojazdów, budowanie ofert wynajmu, obsługę rezerwacji oraz powiązanych z nimi rozliczeń i płatności. Wspiera także terenowe procesy wydania i zwrotu pojazdu. Uzupełnieniem są automatyczna komunikacja z klientami, generowanie dokumentów oraz publiczna strona wypożyczalni, na której klient wypożyczalni przegląda ofertę i składa rezerwacje. Warstwa platformowa obejmuje natomiast subskrypcje i pakiety, administrację wszystkich wypożyczalni oraz komunikację prowadzoną w skali całej platformy.

Dostęp do platformy oferowany jest w postaci pakietów subskrypcyjnych. Wypożyczalnia wybiera odpowiedni pakiet, który określa warunki rozliczenia oraz dopuszczalne limity zasobów, takie jak liczba oddziałów, pojazdów i pracowników. Ograniczenia wynikające z wybranego pakietu są następnie uwzględniane w regułach biznesowych poszczególnych modułów systemu.

Produkt składa się z kilku współpracujących aplikacji dostosowanych do różnych grup odbiorców. Panel wypożyczalni służy właścicielowi oraz pracownikom biurowym do codziennej obsługi działalności, dedykowana aplikacja mobilna wspiera pracowników operacyjnych w procesach terenowych, publiczna strona wypożyczalni udostępnia ofertę klientom wypożyczalni, a panel administracyjny pozwala administratorowi platformy zarządzać wypożyczalniami, pakietami oraz ustawieniami globalnymi. Łącznie funkcjonalność systemu obejmuje kilkanaście obszarów, a jego interfejsy są dostępne w wielu wersjach językowych i obsługują wiele walut. Tak zarysowany obraz produktu wyznacza ramy, w których w kolejnych podrozdziałach przedstawiono aktorów, wymagania oraz model dziedziny systemu.

5.2. Aktorzy systemu

W systemie Wellhicle wyróżniono aktorów reprezentujących użytkowników platformy oraz zewnętrzne systemy zaangażowane w realizację wybranych procesów biznesowych. Ze względu na warstwową strukturę systemu role zostały pogrupowane w cztery warstwy: aktorów globalnych, personel wypożyczalni, klientów wypożyczalni oraz aktorów technicznych.

Podział aktorów jest determinowany przez zakres obowiązków poszczególnych ról oraz miejsce ich występowania w systemie. Umożliwia to wyraźne odróżnienie funkcji zarządczych

platformy od tych skoncentrowanych na obsłudze konkretnej wypożyczalni, a także identyfikację uczestników działających z perspektywy klienta lub integracji technicznej.

5.2.1. Tabela zbiorcza aktorów

Tabela 12 przedstawia wszystkich aktorów systemu oraz ich powiązania z modułami funkcjonalnymi opisanymi w podrozdziale *5.3*. Kolumna „Moduły” wskazuje trzyliterowe skróty modułów (objaśnione w *Tabela 11*), w których dany aktor jest inicjatorem lub odbiorcą co najmniej jednego przypadku użycia; pełna mapa modułów wraz z numerami sekcji znajduje się we wprowadzeniu do podrozdziału *5.3*.

Tabela 12 Podsumowanie wszystkich aktorów systemu. Źródło: opracowanie własne.

Aktor	Grupa	Rola	Moduły
Administrator platformy	Aktorzy globalni	Operator Wellhicle; zarządza platformą oraz wypożyczalniami.	FIN, PLA, PAK, ADM, KOM, SPL
Gość platformy SaaS	Aktorzy globalni	Niezalogowany odwiedzający stronę publiczną Wellhicle. Przegląda ofertę pakietów i rejestruje nową wypożyczalnię.	PAK, SPL
Właściciel wypożyczalni	Personel wypożyczalni	Reprezentuje firmę; jedyny właściciel danej wypożyczalni. Posiada pełne uprawnienia pracownika biurowego i operacyjnego, dodatkowo zarządza pakietem SaaS, opłatami oraz konfiguracją instancji.	ZWY, FLO, KLI, OFE, REZ, FIN, PLA, OPE, PAK, KOM
Pracownik biurowy	Personel wypożyczalni	Operator panelu webowego wypożyczalni. Zarządza flotą, klientami, ofertami i rezerwacjami w imieniu klienta oraz prowadzi obsługę rozliczeń sprzedażowych.	FLO, KLI, OFE, REZ, FIN, PLA
Pracownik operacyjny	Personel wypożyczalni	Wydaje i odbiera pojazdy z udziałem klienta; główny użytkownik aplikacji mobilnej.	OPE
Klient wypożyczalni	Klienci wypożyczalni	Zarejestrowany użytkownik publicznej strony wypożyczalni. Rezerwuje pojazdy, podpisuje elektronicznie umowę najmu, prowadzi historię rezerwacji i ma wgląd we własne dokumenty rozliczeniowe.	KLI, REZ, FIN, PLA, OPE, SWY
Gość wypożyczalni	Klienci wypożyczalni	Niezalogowany odwiedzający publiczną stronę wypożyczalni. Przegląda i filtruje oferty, inicjuje rezerwację oraz rejestruje konto.	KLI, SWY
Bramka płatnicza	Aktorzy techniczni	Zewnętrzny operator płatności obsługujący autoryzację transakcji online i zwroty.	REZ, PLA
System pocztowy	Aktorzy techniczni	Zewnętrzna usługa SMTP odpowiedzialna za dostarczanie wiadomości e-mail (potwierdzenia rezerwacji, reset hasła, powiadomienia administracyjne).	KLI, REZ, PLA, ADM, KOM
System automatyzacji	Aktorzy techniczni	Wewnętrzny mechanizm zdarzeniowo-harmonogramowy uruchamiający akcje bez interwencji użytkownika (anulacje oczekujących rezerwacji, dezaktywacja nieaktywnych instancji, odnawianie subskrypcji, wysyłka powiadomień).	OFE, REZ, FIN, PLA, PAK, KOM, SPL
System ERP	Aktorzy techniczni	Zewnętrzny system księgowy konsumujący dane modułu Finanse i wystawiający po swojej stronie faktury VAT (Wellhicle nie nalicza VAT ani nie generuje numeracji faktur).	FIN

5.2.2. Aktorzy globalni

Aktorzy globalni działają na poziomie całej platformy Wellhicle, niezależnie od konkretnej wypożyczalni – są to administrator reprezentujący operatora platformy oraz gość platformy SaaS jako potencjalny przyszły właściciel wypożyczalni.

Administrator platformy jako kluczowa rola w systemie, sprawuje nadzór nad instancjami wszystkich wypożyczalni (w tym możliwość przejęcia roli dowolnego właściciela w celu wsparcia technicznego), zarządza ofertą pakietów usługowych oraz subskrypcji, prowadzi katalog modeli pojazdów i czarną listę subdomen, konfiguruje dane firmy platformy do fakturowania, nadzoruje

konfigurację bramek płatniczych na poziomie platformy oraz zarządza szablonami dokumentów i wiadomości.

Gość platformy SaaS to niezalogowany użytkownik przeglądający stronę produktową Wellhicle. Ma możliwość zapoznania się z ofertą systemu, przeglądania dostępnych pakietów usługowych i rozpoczęcia procesu rejestracji nowej wypożyczalni. Jego rola kończy się w momencie utworzenia konta firmowego i wyboru pakietu, po czym aktor przechodzi w rolę właściciela wypożyczalni. Gość platformy SaaS nie ma dostępu do instancji wypożyczalni i nie jest tożsamy z gościem na stronie konkretnej wypożyczalni.

5.2.3. Personel wypożyczalni

Personel wypożyczalni działa wyłącznie w obrębie konkretnej wypożyczalni korzystającej z systemu. W tej grupie wyróżniono trzy role: właściciela oraz dwa typy pracowników – pracownika biurowego i pracownika operacyjnego, pracujących na różnych interfejsach (panel zarządzania wypożyczalnią, aplikacja mobilna) i zakresem odpowiedzialności.

Właściciel wypożyczalni dziedziczy uprawnienia po pracowniku biurowym i operacyjnym, co zapewnia mu dostęp do panelu zarządzania wypożyczalnią oraz aplikacji mobilnej. Każda wypożyczalnia posiada dokładnie jednego właściciela odpowiedzialnego za konfigurację swojej instancji, tworzenie oddziałów i zarządzanie pracownikami oraz metodami płatności. W zakres jego obowiązków wchodzi również konfiguracja szablonów powiadomień mailowych, zarządzanie subskrypcją i rozliczeniami finansowymi z platformą Wellhicle. Dostęp właściciela jest ograniczony wyłącznie do jego instancji – nie ma on możliwości przeglądania danych innych wypożyczalni ani korzystania z funkcji administracyjnych przeznaczonych dla administratora platformy.

Pracownik biurowy jest odpowiedzialny za codzienną obsługę wypożyczalni z poziomu panelu zarządzania, gdzie prowadzi ewidencję floty, nadzoruje serwisy pojazdów oraz zarządza ofertami wynajmu. Do jego obowiązków należy również utrzymanie bazy klientów, obsługa rezerwacji składanych za pośrednictwem strony publicznej wypożyczalni oraz prowadzenie rozliczeń sprzedażowych (dane klienta, pozycje dokumentu i anulowanie).

Pracownik operacyjny jest odpowiedzialny za czynności wykonywane przy wydaniu i odbiorze pojazdu, realizowane z użyciem aplikacji mobilnej. Weryfikuje tożsamość klienta, przeprowadza inspekcję stanu technicznego pojazdu oraz dokumentuje jego wygląd i ewentualne uszkodzenia przed wydaniem lub w trakcie zwrotu. Obsługuje również podpis elektroniczny umowy najmu, a także zgłasza usterki stwierdzone po odbiorze auta przez klienta. Zakres kompetencji pracownika operacyjnego jest wyraźnie oddzielony od zakresu obowiązków pracownika biurowego – wspólnym punktem nadrzędnym dla obu ról pozostaje właściciel.

5.2.4. Klienci wypożyczalni

Klienci wypożyczalni korzystają z publicznej strony konkretnej wypożyczalni jako osoby wynajmujące pojazdy. Grupa obejmuje osoby niezalogowane (gości) oraz klientów posiadających konto; przejście z roli gościa do pełnoprawnego klienta następuje przez rejestrację lub logowanie i może nastąpić w dowolnym momencie, w tym podczas konfigurowania rezerwacji.

Gość wypożyczalni to niezalogowany użytkownik korzystający ze strony publicznej konkretnej wypożyczalni. Może przeglądać dostępne Oddziały, filtrować i wyświetlać szczegóły ofert oraz rozpocząć proces rezerwacji poprzez wybór pojazdu, terminu wypożyczenia i ewentualnych dodatków. Ma również możliwość założenia konta klienta. Jego uprawnienia nie obejmują jednak działań wymagających identyfikacji, takich jak finalizacja Rezerwacji, dostęp do historii rezerwacji czy

anulowanie rezerwacji. Aby dokończyć proces Rezerwacji, gość musi przejść na rolę klienta wypożyczalni poprzez logowanie lub rejestrację.

Klient wypożyczalni dziedziczy uprawnienia po gościu wypożyczalni, dlatego zachowuje możliwość przeglądania ofert i konfigurowania rezerwacji. Jako zalogowany użytkownik uzyskuje dodatkowo finalizację rezerwacji z płatnością online, wgląd w historię oraz szczegóły własnych transakcji, możliwość anulowania aktywnych rezerwacji i zarządzanie danymi konta. Przy odbiorze lub zwrocie pojazdu klient składa podpis elektroniczny na umowie najmu oraz protokole zdawczo-odbiorczym; ma także wgląd we własne dokumenty rozliczeniowe z możliwością pobrania ich plików. Dostęp klienta jest ograniczony wyłącznie do jego danych – nie ma on możliwości przeglądania informacji dotyczących innych użytkowników ani danych operacyjnych wypożyczalni.

5.2.5. Aktorzy techniczni

Aktorzy techniczni to systemy współdziałające z platformą Wellhicle bez udziału człowieka. Wyróżniono wśród nich zewnętrzne usługi, takie jak bramka płatnicza, system pocztowy czy system ERP oraz wewnętrzny mechanizm automatyzacji.

Bramka płatnicza to zewnętrzny operator płatności wykorzystywany przy obsłudze transakcji online; odpowiada za autoryzację płatności oraz obsługę zwrotów. W systemie występuje w procesie potwierdzania rezerwacji z płatnością online oraz w module Płatności (transakcje, konfiguracja metod).

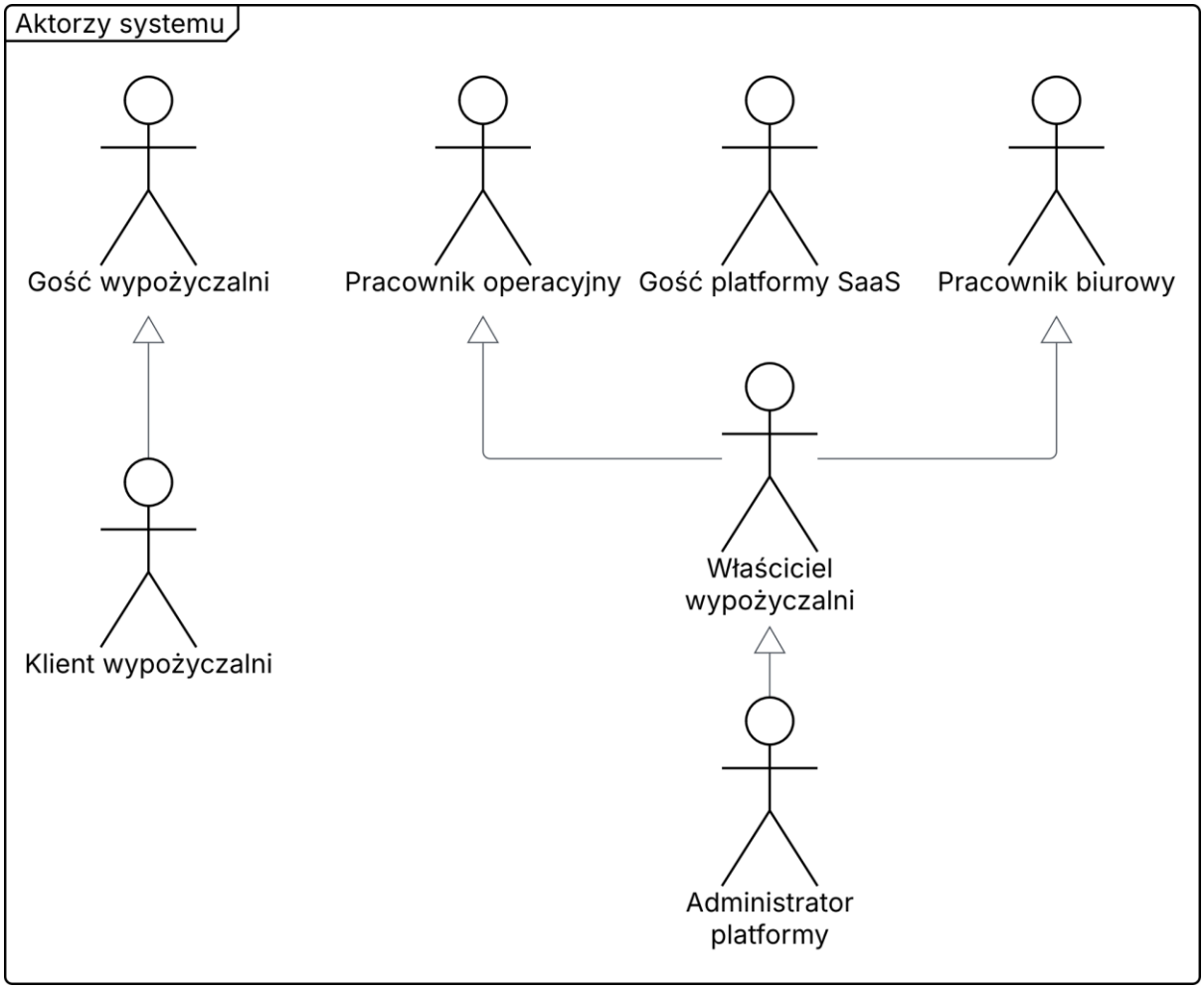
System pocztowy to zewnętrzna usługa SMTP wykorzystywana przez platformę Wellhicle do wysyłki wiadomości e-mail w komunikacji z klientami oraz właścicielami wypożyczalni. W zależności od konfiguracji danej wypożyczalni, serwerem nadawczym może być wybrany przez właściciela zewnętrzny dostawca usług pocztowych lub domyślne konto domenowe Wellhicle.

System automatyzacji to wewnętrzny mechanizm realizujący działania wykonywane w sposób zautomatyzowany i cykliczny – reaguje na zdarzenia systemowe oraz uruchamia zaplanowane zadania według harmonogramu. Obejmuje między innymi anulowanie nieopłaconych rezerwacji, dezaktywację nieaktywnych instancji wypożyczalni, odnawianie subskrypcji, wysyłkę przypomnień oraz aktualizację statusów pojazdów.

System ERP to zewnętrzny system księgowy, który konsumuje dane modułu Finanse. Zakres integracji obejmuje przekazywanie danych fakturowo-rozliczeniowych – faktura VAT jest wystawiana wyłącznie przez system ERP, a sama platforma nie nalicza podatku VAT ani nie generuje numeracji faktur.

5.2.6. Dziedziczenie aktorów

Hierarchia dziedziczenia aktorów została przedstawiona na *Rysunek 16* i obejmuje wyłącznie role użytkowników systemu, z pominięciem aktorów technicznych. Aktorzy techniczni nie uczestniczą w dziedziczeniu – współpracują z systemem jedynie w określonych obszarach funkcjonalnych (płatności, komunikacja, automatyzacja).



Rysunek 16 Hierarchia dziedziczenia aktorów systemu Wellhicle. Źródło: opracowanie własne

Klient wypożyczalni dziedziczy po gościu wypożyczalni, co oznacza, że może korzystać zarówno z funkcji dostępnych dla użytkownika niezalogowanego, jak i tych wymagających konta. Właściciel dziedziczy po pracowniku biurowym oraz operacyjnym, łącząc tym samym zakres uprawnień dostępnych zarówno w panelu webowym, jak i aplikacji mobilnej. Administrator platformy dziedziczy po właścicielu, co umożliwia mu przejęcie jego roli podczas wsparcia technicznego. Gość platformy SaaS nie uczestniczy w hierarchii dziedziczenia, ponieważ jego rola ogranicza się wyłącznie do interakcji z częścią produktową systemu – nie przejmuje uprawnień żadnego innego aktora ani nie przekazuje własnych uprawnień innym rolom.

5.3. Zakres funkcjonalny

Niniejsza sekcja przedstawia biznesową warstwę wymagań systemu, koncentrując się na opisie funkcjonalności niezależnie od technologii i implementacji. Struktura podziału na trzynaście modułów odzwierciedla granice decyzyjne oraz obszary własności produktu: pierwsze dziewięć modułów (*5.3.1-5.3.9*) tworzy operacyjny rdzeń systemu, obejmujący konfigurację instancji, zarządzanie flotą, klientami, ofertami i rezerwacjami, rozliczenia oraz procesy wydania i zwrotu pojazdu wraz z publiczną stroną wypożyczalni. Cztery ostatnie moduły (*5.3.10-5.3.13*) opisują warstwę platformową, tj. subskrypcję pakietów dla wypożyczalni, administrację globalną, komunikację e-mail oraz publiczną stronę samej platformy Wellhicle.

Wymagania funkcjonalne przedstawione w tej sekcji stanowią podstawę dla pozostałych artefaktów analizy: przypadki użycia (podrozdział *5.5*) opisują wybrane wymagania za pomocą

scenariuszy interakcji aktora z systemem, diagramy aktywności (podrozdział *5.6*) ilustrują przepływy procesów angażujących wiele modułów, analityczny model dziedziny (podrozdział *5.7*) dostarcza atrybutów dla encji wymienionych w wymaganiach, natomiast wymagania niefunkcjonalne (podrozdział *5.4*) uzupełniają całość o aspekty jakościowe i przekrojowe – wydajność, bezpieczeństwo, zgodność oraz użyteczność systemu.

Każdy moduł rozpoczyna się od krótkiego opisu jego roli w systemie, po którym następuje tabela określająca obecność poszczególnych aktorów z katalogu przedstawionego w podrozdziale *5.2* oraz ich role w danym module. Kolejnym elementem jest lista wymagań funkcjonalnych pogrupowanych tematycznie, następnie tabela reguł biznesowych i walidacji, a całość zamyka zestawienie powiązań z innymi modułami.

5.3.1. Zarządzanie wypożyczalnią

Moduł Zarządzanie wypożyczalnią pełni rolę nadrzędnego modułu konfiguracyjnego instancji Wypożyczalni. Umożliwia on właścicielowi kompleksową administrację techniczną, biznesową i wizualną swojej instancji oraz definiuje parametry i polityki wykorzystywane przez pozostałe moduły operacyjne. Każda wypożyczalnia funkcjonuje niezależnie, posiadając odrębne dane i konfigurację.

5.3.1.1. Aktorzy modułu

Tabela 13 Aktorzy modułu Zarządzanie wypożyczalnią. Źródło: opracowanie własne.

Aktor	Rola w module
Właściciel wypożyczalni	Przechodzi przez kreator pierwszej konfiguracji i prowadzi bieżącą administrację instancją wypożyczalni.

5.3.1.2. Wymagania funkcjonalne

Kreator pierwszej konfiguracji:

- **WF-ZWY-01:** Podczas pierwszego logowania właściciela uruchamiany jest kreator konfiguracji wypożyczalni; bez ukończenia tego procesu dalsza praca w panelu nie jest możliwa.
- **WF-ZWY-02:** W ramach kreatora właściciel wprowadza dane identyfikacyjne firmy, parametry operacyjne wypożyczalni (subdomena, waluta główna, obsługiwane waluty, język główny oraz obsługiwane języki), a także wybiera pakiet subskrypcji i metodę płatności.
- **WF-ZWY-03:** Po zatwierdzeniu ustawień w kreatorze tworzona jest aktywna instancja wypożyczalni, a właściciel zostaje przekierowany do widoku głównego panelu administracyjnego.

Waluta główna:

- **WF-ZWY-04:** Wszystkie ceny ofert, dodatków i płatności są domyślnie prezentowane w walucie głównej wypożyczalni.
- **WF-ZWY-05:** Właściciel może zmienić walutę główną wypożyczalni; taka zmiana nie powoduje przeliczenia istniejących cen ofert, dodatków ani płatności.

Wygląd publicznej strony wypożyczalni:

- **WF-ZWY-06:** Właściciel konfiguruje wygląd publicznej strony wypożyczalni, określając kolorystykę z predefiniowanej palety barw oraz krój czcionki spośród dostępnych opcji. Może również przesłać logo i zdjęcie nagłówka strony, a także edytować treść sekcji: nagłówkowej,

kontaktowej oraz stopki – w językach obsługiwanych przez daną wypożyczalnię. Dodatkowo konfiguruje linki do profili społecznościowych (Facebook, Instagram, TikTok).

Polityka prywatności i regulamin serwisu:

- **WF-ZWY-07:** Właściciel wgrywa własne dokumenty polityki prywatności oraz regulaminu serwisu w postaci plików PDF; oba dokumenty mogą zostać w dowolnym momencie podmienione.
- **WF-ZWY-08:** Aktywacja publicznej strony wypożyczalni dla klientów wypożyczalni wymaga obecności aktualnych wersji obu dokumentów; po aktywacji są one dostępne publicznie ze strony wypożyczalni.

Subdomena i własna domena:

- **WF-ZWY-09:** W kreatorze właściciel wybiera subdomenę przypisywaną do wypożyczalni.
- **WF-ZWY-10:** Właściciel podłącza, zmienia lub odłącza zewnętrzną domenę dla swojej wypożyczalni.
- **WF-ZWY-11:** System udostępnia instrukcję konfiguracji DNS oraz monitoruje poprawność rekordów; do momentu ich potwierdzenia status domeny pozostaje niezweryfikowany.

Oddziały:

- **WF-ZWY-12:** Właściciel tworzy, edytuje i usuwa oddziały wypożyczalni. Każdy oddział posiada indywidualne dane identyfikacyjne (nazwę, adres oraz informacje kontaktowe), przypisaną strefę czasową, określone godziny otwarcia i konfigurowalny okres karencji dla nieodebranych rezerwacji.

Pracownicy:

- **WF-ZWY-13:** Właściciel tworzy konta pracowników biurowych oraz operacyjnych i zarządza nimi.
- **WF-ZWY-14:** Na koncie pracownika dostępne są następujące operacje: reset hasła, dezaktywacja konta, ponowna aktywacja wcześniej dezaktywowanego konta, zmiana przypisanego oddziału oraz trwałe usunięcie konta po uprzedniej dezaktywacji.
- **WF-ZWY-15:** Po dezaktywacji konta pracownika lub dezaktywacji macierzystego oddziału wszystkie aktywne sesje użytkownika zostają natychmiast unieważnione, a dostęp do systemu zostaje niezwłocznie cofnięty.

Szablony dokumentów:

- **WF-ZWY-16:** Właściciel wgrywa lub podmienia szablony dokumentów generowanych przez system (umowa najmu, protokół zdawczo-odbiorczy).
- **WF-ZWY-17:** Przy edycji szablonu system prezentuje listę pól dostępnych dla danego typu dokumentu.

Dezaktywacja i przywracanie konta wypożyczalni:

- **WF-ZWY-18:** Właściciel dezaktywuje konto wypożyczalni; proces ten blokuje możliwość logowania dla wszystkich pracowników i klientów oraz unieważnia ich aktywne sesje, jednocześnie zachowując pełną integralność danych należących do tej wypożyczalni.
- **WF-ZWY-19:** Właściciel ma możliwość przywrócenia dezaktywowanego konta wypożyczalni, pod warunkiem posiadania aktywnej subskrypcji odpowiedniego pakietu usług.

Jednostka przebiegu pojazdów:

- **WF-ZWY-20:** Właściciel zmienia jednostkę przebiegu pojazdów wypożyczalni (kilometry albo mile); jednostka obowiązuje przy prezentacji przebiegu pojazdów i zapisach stanu technicznego we wszystkich modułach operacyjnych.

5.3.1.3. Reguły biznesowe i walidacje

Tabela 14 Reguły biznesowe i walidacje modułu Zarządzanie wypożyczalnią. Źródło: opracowanie własne.

Nr	Reguła
RB-ZWY-01	Każda wypożyczalnia ma dokładnie jednego właściciela.
RB-ZWY-02	Każda wypożyczalnia posiada wyłącznie jedną walutę główną, określoną kodem ISO 4217 oraz jeden język główny.
RB-ZWY-03	Zmiana ustawionej waluty głównej nie powoduje automatycznego przeliczenia istniejących cen ofert, dodatków ani kwot płatności.
RB-ZWY-04	Subdomena wypożyczalni musi być unikalna na poziomie całej platformy.
RB-ZWY-05	Wybrana subdomena nie może znajdować się na liście zastrzeżonych subdomen prowadzonej przez administratora platformy.
RB-ZWY-06	Aktywacja publicznej strony wypożyczalni jest możliwa tylko po uprzednim dodaniu dokumentów polityki prywatności i regulaminu serwisu.
RB-ZWY-07	Wypożyczalnia musi posiadać co najmniej jeden aktywny oddział, aby móc przyjmować rezerwacje pojazdów.
RB-ZWY-08	Liczba oddziałów, pojazdów oraz pracowników (biurowych i operacyjnych) nie może przekraczać limitów określonych w aktualnej subskrypcji.
RB-ZWY-09	Obniżenie planu abonamentowego na wariant o niższych limitach nie usuwa istniejących zasobów, ale blokuje możliwość dodawania nowych elementów powyżej progu nowego pakietu.
RB-ZWY-10	Dezaktywacja oddziału jest dopuszczalna tylko wtedy, gdy nie są do niego przypisane żadne aktywne pojazdy ani oferty wynajmu.
RB-ZWY-11	Trwałe usunięcie oddziału wymaga uprzedniej dezaktywacji oraz braku jakichkolwiek przypisanych pracowników lub pojazdów.
RB-ZWY-12	Dezaktywacja konta pracownika lub jego macierzystego oddziału powoduje natychmiastowe wygaśnięcie wszystkich aktywnych sesji użytkownika.
RB-ZWY-13	Adres e-mail przypisany do pracownika musi być unikalny w skali całej platformy.
RB-ZWY-14	Trwałe usunięcie konta pracowniczego jest możliwe wyłącznie po wcześniejszej dezaktywacji konta; konto właściciela wypożyczalni nie podlega trwałemu usunięciu.
RB-ZWY-15	Dezaktywacja wypożyczalni unieważnia sesje logowania wszystkich pracowników i klientów, blokując możliwość dalszego logowania, ale zachowuje dane konfiguracyjne wypożyczalni.

5.3.1.4. Powiązania z innymi modułami

Moduł Zarządzanie wypożyczalnią jest nadrzędny wobec pozostałych modułów operacyjnych – udostępnia im konfigurację instancji.

Tabela 15 Powiązania modułu Zarządzanie wypożyczalnią z innymi modułami systemu. Źródło: opracowanie własne.

Moduł	Zakres powiązania
Flota	Pojazdy są przypisywane do oddziałów zdefiniowanych w tym module.
Oferty	Każda oferta jest przypisana do określonego oddziału, a jej cennik obowiązkowo zawiera cenę wyrażoną w walucie głównej danej wypożyczalni.
Rezerwacje	Rezerwacje są obsługiwane w kontekście konkretnego oddziału; szablony umowy najmu oraz protokołu zdawczo-odbiorczego pochodzą z tego modułu.
Płatności	Waluta główna i obsługiwane waluty wypożyczalni (zdefiniowane w tym module) wyznaczają dopuszczalne opcje rozliczeń; sama konfiguracja metod płatności jest przechowywana osobno.
Strona publiczna wypożyczalni	Wygląd strony, treść polityki prywatności i regulaminu, dane kontaktowe oraz adres (subdomena lub domena) pochodzą z tego modułu.
Komunikacja	Szablony wiadomości stanowią źródło dla powiadomień wysyłanych do klientów, gości lub innych użytkowników.
Finanse	Waluta główna wypożyczalni determinuje walutę rachunków sprzedażowych; oddział, w którym utworzono rezerwację, jest automatycznie przypisywany jako powiązanie do odpowiedniego rozliczenia.

5.3.2. Flota

Moduł Flota jest odpowiedzialny za pełen cykl życia pojazdów wypożyczalni – począwszy od ich rejestracji w systemie, przez przypisanie do oddziałów, obsługę serwisów technicznych i ręczne blokady dostępności, aż po trwałe usunięcie. Stan każdego pojazdu (aktywny, nieaktywny, w wynajmie, w serwisie lub zablokowany) stanowi niezmiennik dla pozostałych modułów operacyjnych – determinuje widoczność pojazdu w ofertach oraz możliwość realizacji rezerwacji. Wartości przebiegu są prezentowane w jednostce ustawionej dla danej wypożyczalni (kilometry lub mile), konfigurowanej w punkcie *5.3.1*.

5.3.2.1. Aktorzy modułu

Tabela 16 Aktorzy modułu Flota. Źródło: opracowanie własne.

Aktor	Rola w module
Właściciel wypożyczalni	Posiada dostęp do floty we wszystkich oddziałach; zarządza pojazdami, serwisami, grupami pojazdów oraz ręcznymi blokadami. Jest jedynym aktorem uprawnionym do operacji przypisywania i przenoszenia pojazdów między oddziałami: dezaktywacji w danym oddziale, ponownej aktywacji oraz transferu pomiędzy lokalizacjami.
Pracownik biurowy	Zarządza flotą wewnątrz swojego oddziału: dodaje, edytuje i usuwa pojazdy, prowadzi serwisy oraz nakłada lub znosi ręczne blokady dostępności. Może także modyfikować przynależność pojazdów do istniejących grup.

5.3.2.2. Wymagania funkcjonalne

Zarządzanie pojazdami:

- **WF-FLO-01:** Pracownik biurowy lub właściciel dodaje pojazd do floty, podając dane techniczne (model, rok produkcji, numer rejestracyjny, VIN, typ nadwozia, rodzaj silnika i skrzynię biegów), opcjonalnie – w obsługiwanych językach wypożyczalni – dodaje również opis oraz zdjęcia. Określa także okres przypisania pojazdu do oddziału (datę rozpoczęcia i ewentualnie zakończenia). Pojazd wprowadzony przez pracownika biurowego automatycznie trafia do jego macierzystego oddziału, natomiast pojazd dodany przez właściciela jest przypisywany do wybranego przez niego oddziału.

- **WF-FLO-02:** Pracownik biurowy edytuje dane techniczne pojazdu oraz zarządza jego zdjęciami – może dodawać, usuwać i oznaczać główne zdjęcie.
- **WF-FLO-03:** Pracownik biurowy trwale usuwa pojazd z floty.

Aktywność pojazdu w oddziale:

- **WF-FLO-04:** Właściciel zarządza aktywnością pojazdu w oddziale – dezaktywuje go, zamykając bieżące przypisanie do oddziału poprzez określenie daty końcowej. Następnie może ponownie aktywować ten sam pojazd przez utworzenie nowego powiązania z lokalizacją; w stanie dezaktywacji nie jest on dostępny dla tworzenia ofert ani rezerwacji.

Zarządzanie serwisami pojazdów:

- **WF-FLO-05:** Pracownik biurowy dodaje serwis pojazdu, określając zakres prac oraz datę rozpoczęcia i opcjonalnie zakończenia (brak tej daty oznacza trwanie do odwołania). Do serwisu można dodać zdjęcia dokumentujące stan pojazdu.
- **WF-FLO-06:** Pracownik biurowy edytuje opis i okres trwania istniejącego serwisu.
- **WF-FLO-07:** Pracownik biurowy kończy lub usuwa aktywny serwis; po zakończeniu pojazd odzyskuje status umożliwiający realizację ofert i rezerwacji.

Przenoszenie pojazdów między oddziałami:

- **WF-FLO-08:** Właściciel przenosi pojazd do innego oddziału, tworząc nowe przypisanie pojazdu do docelowego oddziału; system zamyka poprzednie aktywne przypisanie z dniem poprzedzającym przekazanie.

Ręczna blokada dostępności:

- **WF-FLO-09:** Pracownik biurowy zarządza ręczną blokadą dostępności pojazdu – nakłada blokadę na wskazany okres (z opcjonalną datą zakończenia, np. na czas wewnętrznych prac organizacyjnych), edytuje trwającą blokadę oraz ją usuwa.

Zarządzanie grupami pojazdów:

- **WF-FLO-10:** Właściciel zarządza grupami pojazdów – tworzy nowe grupy, określając ich nazwę i opis w językach obsługiwanych przez wypożyczalnię, dodaje zdjęcia (w tym główne) oraz początkowy skład pojazdów. Może również edytować istniejące grupy lub trwale je usuwać.
- **WF-FLO-11:** Pracownik biurowy modyfikuje przynależność konkretnego pojazdu do aktywnych grup – dodaje go do wybranych grup lub usuwa z nich.

Wgląd w dostępność pojazdu:

- **WF-FLO-12:** Pracownik biurowy przegląda kalendarz dostępności pojazdu, agregujący rezerwacje, serwisy oraz ręczne blokady przypisane do pojazdu we wskazanym oknie czasowym.

5.3.2.3. Reguły biznesowe i walidacje

Tabela 17 Reguły biznesowe i walidacje modułu Flota. Źródło: opracowanie własne.

Nr	Reguła
RB-FLO-01	Każdy pojazd posiada w danym momencie dokładnie jedno aktywne przypisanie do oddziału.
RB-FLO-02	Okresy wykorzystania pojazdu (rezerwacje, serwisy, ręczne blokady) w ramach jednego przypisania do oddziału nie mogą się nakładać.
RB-FLO-03	Każde wykorzystanie pojazdu musi mieścić się w okresie aktywnego przypisania go do danego oddziału.

Nr	Reguła
RB-FLO-04	Liczba pojazdów należących do danej wypożyczalni nie może przekraczać limitu ustalonego dla jej aktualnej subskrypcji.
RB-FLO-05	Dezaktywacja pojazdu w oddziale jest dopuszczalna tylko pod warunkiem, że na okres przypadający po dacie dezaktywacji nie są zaplanowane żadne aktywne rezerwacje ani trwające serwisy.
RB-FLO-06	Trwałe usunięcie pojazdu jest możliwe wyłącznie wtedy, gdy nigdy nie był on użyty w rezerwacji, serwisie lub ręcznej blokadzie oraz nie posiada powiązanych ofert.
RB-FLO-07	Przeniesienie pojazdu między oddziałami jest dopuszczalne tylko wtedy, gdy w okresie następującym po dacie przekazania nie pozostają aktywne żadne wykorzystania pojazdu w poprzednim oddziale.
RB-FLO-08	Przeniesienie pojazdu między oddziałami powoduje dezaktywację ofert związanych z tym pojazdem w dotychczasowym oddziale.
RB-FLO-09	Dezaktywacja pojazdu w danym oddziale lub rozpoczęcie serwisu skutkuje wyłączeniem możliwości rezerwacji tego pojazdu poprzez oferty powiązane z nim.
RB-FLO-10	Aktywacja oferty pojazdu jest dopuszczalna jedynie wtedy, gdy sam pojazd posiada status „aktywny”.
RB-FLO-11	Pojazd może być przypisany do wielu grup pojazdów równocześnie.
RB-FLO-12	Trwałe usunięcie grupy pojazdów jest dopuszczalne wyłącznie w sytuacji, gdy żadna oferta nie jest powiązana z tą grupą.
RB-FLO-13	Numer VIN pojazdu musi być unikalny w ramach danej wypożyczalni.

5.3.2.4. Powiązania z innymi modułami

Moduł Flota jest źródłem prawdy o pojazdach i ich dostępności dla pozostałych modułów operacyjnych.

Tabela 18 Powiązania modułu Flota z innymi modułami systemu. Źródło: opracowanie własne.

Moduł	Zakres powiązania
Oferty	Oferta jest tworzona na konkretny pojazd lub grupę pojazdów; bieżący stan pojazdu oraz aktywne przypisanie do oddziału określają niezmiennik jej dostępności.
Rezerwacje	Rezerwacja dotyczy konkretnego pojazdu i musi mieścić się w ramach jego aktualnego przydziału do oddziału; trwający serwis lub ręczna blokada uniemożliwiają utworzenie nowej rezerwacji.
Operacyjny	Operacje wydania i odbioru pojazdu oraz protokoły stanu technicznego operują na pojazdach; stwierdzenie usterki podczas zwrotu skutkuje automatycznym utworzeniem zgłoszenia serwisowego.

5.3.3. Klienci

Moduł Klienci odpowiada za zarządzanie danymi klientów wypożyczalni oraz ich kontami. Umożliwia samodzielną rejestrację i logowanie przez stronę publiczną, a także pełną obsługę profili po stronie pracowników biurowych. Każdy klient ma dwa rozdzielne wymiary aktywności: profil (decydujący o możliwości uczestnictwa w rezerwacjach) oraz konto (określające uprawnienia do logowania), co pozwala na obsługę klientów bez kont online i precyzyjne zarządzanie danymi osobowymi. Moduł wykrywa duplikaty adresów e-mail w obrębie jednej wypożyczalni oraz umożliwia nieodwracalną anonimizację danych osobowych.

5.3.3.1. Aktorzy modułu

Tabela 19 Aktorzy modułu Klienci. Źródło: opracowanie własne.

Aktor	Rola w module
Klient wypożyczalni	Rejestruje konto przez stronę publiczną wypożyczalni, loguje się i samodzielnie aktualizuje swoje dane osobowe, kontaktowe oraz dokument tożsamości.
Pracownik biurowy	Tworzy profile klientów (również tych bez konta online), edytuje ich dane i zarządza stanem profili oraz kont – dezaktywuje je i ponownie aktywuje.
Właściciel wypożyczalni	Wykonuje wszystkie operacje pracownika biurowego oraz jest jedynym aktorem uprawnionym do anonimizacji klienta i jego trwałego usunięcia.

5.3.3.2. Wymagania funkcjonalne

Rejestracja i logowanie klienta:

- **WF-KLI-01:** Klient rejestruje konto na stronie publicznej wypożyczalni, podając adres e-mail, hasło, imię, nazwisko oraz numer telefonu i preferowany język; dodatkowo może wprowadzić dane takie jak: adres zamieszkania, informacje o dokumencie tożsamości lub szczegóły firmy.
- **WF-KLI-02:** Po zakończeniu rejestracji system automatycznie wysyła na podany adres e-mail wiadomość zawierającą link aktywacyjny; konto staje się aktywne dopiero po kliknięciu tego linku.
- **WF-KLI-03:** Jeśli klient próbuje zarejestrować konto na adres e-mail powiązany już z istniejącym profilem utworzonym wcześniej przez pracownika biurowego, system prosi o potwierdzenie tożsamości poprzez podanie imienia, nazwiska i numeru telefonu odpowiadających danym profilu; rejestracja zostaje odrzucona w przypadku niespójności danych.

Edycja danych przez klienta:

- **WF-KLI-04:** Zalogowany klient aktualizuje swoje dane, takie jak imię, nazwisko, numer telefonu, język preferowany, adres zamieszkania oraz dokument tożsamości; dodatkowo może wprowadzić lub zmodyfikować dane firmy.

Obsługa klientów przez pracowników wypożyczalni:

- **WF-KLI-05:** Pracownik biurowy tworzy profil klienta bez konta online, podając dane identyfikacyjne i kontaktowe; utworzony w ten sposób klient może później dołączyć konto poprzez rejestrację na ten sam adres e-mail.
- **WF-KLI-06:** Pracownik biurowy edytuje dane klienta z aktywnym profilem.

Aktywność profilu i konta klienta:

- **WF-KLI-07:** Pracownik biurowy zarządza aktywnością profilu klienta – dezaktywuje go, automatycznie unieważniając wszystkie aktywne sesje i blokując dostęp do konta. Może również ponownie aktywować uprzednio zdezaktywowany profil.
- **WF-KLI-08:** Pracownik biurowy dezaktywuje samo konto klienta, co skutkuje uniemożliwieniem logowania i wymusza ponowną aktywację konta przed kolejnym dostępem – nie wpływa to jednak na status profilu.
- **WF-KLI-09:** Pracownik biurowy wylogowuje klienta z aktywnej sesji konta, co skutkuje natychmiastowym unieważnieniem bieżącej sesji i wymuszeniem ponownego logowania – nie wpływa to jednak ani na status profilu, ani na status konta klienta.

Anonimizacja klienta:

- **WF-KLI-10:** Właściciel anonimizuje klienta z dezaktywowanym profilem i kontem; operacja zastępuje dane osobowe wartościami neutralnymi i jest nieodwracalna.

5.3.3.3. Reguły biznesowe i walidacje

Tabela 20 Reguły biznesowe i walidacje modułu Klienci. Źródło: opracowanie własne.

Nr	Reguła
RB-KLI-01	Adres e-mail klienta jest unikalny w obrębie wypożyczalni.
RB-KLI-02	Rejestracja konta klienta wymaga akceptacji regulaminu serwisu oraz polityki prywatności obowiązujących w danej wypożyczalni.
RB-KLI-03	Dołączenie nowego konta do istniejącego profilu klienta jest możliwe wyłącznie, gdy podane przy rejestracji imię, nazwisko i numer telefonu są zgodne z danymi w tym profilu.
RB-KLI-04	Aktywacja konta klienta wymaga istnienia aktywnego profilu przypisanego do tego konta.
RB-KLI-05	Dezaktywacja profilu klienta powoduje automatyczną dezaktywację jego konta oraz unieważnienie wszystkich aktywnych sesji logowania.
RB-KLI-06	Dezaktywacja konta klienta blokuje możliwość logowania i tworzenia nowych rezerwacji, ale nie wpływa na istniejące profile (jeśli są aktywne), wcześniej utworzone rezerwacje, rozliczenia ani dokumenty.
RB-KLI-07	Klient z dezaktywowanym profilem lub poddany anonimizacji nie może być dodany do nowej rezerwacji ani powiązany z nowym rozliczeniem.
RB-KLI-08	Edycja danych klienta, który został zanonimizowany, jest niedozwolona.
RB-KLI-09	Anonimizację klienta można przeprowadzić tylko wtedy, gdy jego profil i konto są nieaktywne.
RB-KLI-10	Anonimizacja klienta jest procesem nieodwracalnym – dane osobowe zostają zastąpione neutralnymi wartościami, a przypisany adres e-mail staje się ponownie dostępny do rejestracji.
RB-KLI-11	Anonimizacja nie modyfikuje istniejących rezerwacji, rozliczeń, płatności ani dokumentów – kopie danych osobowych w tych artefaktach pozostają nienaruszone dla zachowania ich mocy prawnej.
RB-KLI-12	Trwałe usunięcie klienta jest możliwe wyłącznie przy nieaktywnym profilu i koncie, bez wpływu na istniejące rezerwacje czy dokumenty powiązane z tym użytkownikiem.

5.3.3.4. Powiązania z innymi modułami

Moduł Klienci dostarcza dane osobowe i kontaktowe wykorzystywane w procesach rezerwacji, rozliczeń i komunikacji.

Tabela 21 Powiązania modułu Klienci z innymi modułami systemu. Źródło: opracowanie własne.

Moduł	Zakres powiązania
Rezerwacje	Każda rezerwacja jest tworzona w imieniu klienta; aktywny profil klienta oraz brak anonimizacji są niezmiennikami utworzenia rezerwacji.
Płatności	Rozliczenie za rezerwację jest powiązane z klientem; jego dane osobowe są wykorzystywane przy realizacji płatności i wystawianiu dokumentów rozliczeniowych.
Komunikacja	Wiadomości dotyczące konta klienta (potwierdzenie rejestracji, weryfikacja adresu e-mail, reset hasła, zmiana statusu konta) są obsługiwane przez moduł komunikacji.

5.3.4. Oferty

Moduł Oferty umożliwia definiowanie warunków najmu, na jakich pojazdy są udostępniane klientom wypożyczalni. Każda oferta łączy konkretny pojazd lub grupę pojazdów z parametrami

komercyjnymi – cennikiem (w jednej lub wielu walutach), dziennym limitem przebiegu, polityką zwrotów oraz dodatkowymi opcjami i okresem obowiązywania. Oferty wyznaczają ramy dla tworzenia rezerwacji przez klientów – aktywne oferty są jedynym źródłem informacji prezentowanych na publicznej stronie wypożyczalni. Dodatki, takie jak fotelik dziecięcy czy ubezpieczenie, zarządzane jako niezależne byty wewnątrz wypożyczalni, mogą być przypisywane do wybranych ofert.

5.3.4.1. Aktorzy modułu

Tabela 22 Aktorzy modułu Oferty. Źródło: opracowanie własne.

Aktor	Rola w module
Pracownik biurowy	Tworzy, edytuje, aktywuje, dezaktywuje i usuwa oferty oraz dodatki w obrębie swojego oddziału; definiuje cenniki ofert i dodatków.
Właściciel wypożyczalni	Wykonuje wszystkie operacje pracownika biurowego dla ofert i dodatków we wszystkich oddziałach wypożyczalni; tworzy oferty grupowe oparte na grupach pojazdów.
System automatyzacji	Dezaktywuje oferty, których okres obowiązywania (data zakończenia) upłynął.

5.3.4.2. Wymagania funkcjonalne

Zarządzanie ofertami pojazdów:

- **WF-OFE-01:** Pracownik biurowy tworzy ofertę dla wybranego pojazdu z aktywnym przypisaniem do oddziału, podając cennik w jednej lub wielu walutach (z obowiązkową ceną w walucie głównej wypożyczalni), dzienny limit przebiegu, kwotę i walutę depozytu, procent zwrotu przy anulowaniu z winy klienta, opcjonalny okres obowiązywania (datę rozpoczęcia oraz opcjonalną datę zakończenia) oraz listę dodatków przypisanych do oferty.
- **WF-OFE-02:** Pracownik biurowy lub właściciel tworzy ofertę dla wybranej grupy pojazdów; oferta utworzona przez pracownika biurowego trafia do oddziału, w którym jest przypisany, natomiast oferta stworzona przez właściciela zostaje przypisana do wskazanego przez niego oddziału. Pozostałe parametry oferty są analogiczne jak dla oferty pojazdowej.
- **WF-OFE-03:** Pracownik biurowy edytuje parametry oferty: cennik, limit przebiegu, depozyt, procent zwrotu przy anulowaniu z winy klienta, okres obowiązywania (datę rozpoczęcia oraz opcjonalną datę zakończenia), listę dodatków i nazwę handlową.
- **WF-OFE-04:** Pracownik biurowy zarządza cyklem życia oferty: aktywuje ją, dezaktywuje oraz trwale usuwa.

Zarządzanie dodatkami:

- **WF-OFE-05:** Pracownik biurowy tworzy dodatek, podając jego nazwę i opis w obsługiwanych językach wypożyczalni, cennik w jednej lub wielu walutach, typ rozliczania (jednorazowo albo za każdy dzień najmu), maksymalną ilość możliwą do zamówienia w pojedynczej rezerwacji oraz procent zwrotu przy anulowaniu.
- **WF-OFE-06:** Pracownik biurowy edytuje parametry dodatku.
- **WF-OFE-07:** Pracownik biurowy zarządza cyklem życia dodatku – aktywuje go, dezaktywuje oraz trwale usuwa.

Automatyczne zarządzanie cyklem życia:

- **WF-OFE-08:** System automatyzacji dezaktywuje oferty, których data zakończenia okresu obowiązywania minęła; oferta dezaktywowana automatycznie nie jest już publikowana na stronie publicznej wypożyczalni i nie może być użyta do utworzenia nowej rezerwacji.

Prezentacja ofert klientom:

- **WF-OFE-09:** System publikuje aktywne oferty na stronie publicznej wypożyczalni; gość oraz klient mogą filtrować je po oddziale, parametrach pojazdu, przedziale cenowym i okresie najmu.

5.3.4.3. Reguły biznesowe i walidacje

Tabela 23 Reguły biznesowe i walidacje modułu Oferty. Źródło: opracowanie własne.

Nr	Reguła
RB-OFE-01	Oferta jest powiązana dokładnie z jednym pojazdem albo dokładnie z jedną grupą pojazdów; obydwa powiązania jednocześnie nie są dopuszczalne.
RB-OFE-02	Oferta musi być przypisana do konkretnego, aktywnego oddziału wypożyczalni.
RB-OFE-03	W obrębie jednego aktywnego przypisania pojazdu do oddziału okresy obowiązywania jego aktywnych ofert nie mogą się nakładać.
RB-OFE-04	Okres obowiązywania oferty musi mieścić się w zakresie czasowym aktywnego przypisania powiązanego pojazdu do oddziału.
RB-OFE-05	Cennik oferty oraz cennik dodatku obowiązkowo zawierają cenę wyrażoną w walucie głównej wypożyczalni; pozostałe waluty są opcjonalne i muszą pochodzić z listy obsługiwanych przez daną wypożyczalnię.
RB-OFE-06	Procent zwrotu oferty oraz procent zwrotu dodatku przyjmują wartości od 0% do 100%.
RB-OFE-07	Aktywacja oferty jest możliwa tylko wtedy, gdy powiązany pojazd ma status „aktywny”, a wszystkie dodatki przypisane do tej oferty są aktywne.
RB-OFE-08	Oferta z wygasłym okresem obowiązywania (data zakończenia przed bieżącym dniem) nie może pozostawać w stanie „aktywna”.
RB-OFE-09	Aktywna oferta jest publikowana na publicznej stronie internetowej wypożyczalni; po dezaktywacji przestaje być widoczna i nie może zostać użyta do utworzenia nowej rezerwacji.
RB-OFE-10	Parametry oferty (cennik, limit przebiegu, procent zwrotu, lista dodatków oraz kwota depozytu) są automatycznie kopiowane do rezerwacji w momencie jej tworzenia i pozostają niezmienne nawet po późniejszej edycji lub usunięciu oferty.
RB-OFE-11	Trwałe usunięcie oferty jest możliwe wyłącznie wtedy, gdy oferta ma status „nieaktywna” i nie istnieją żadne rezerwacje utworzone na jej podstawie.
RB-OFE-12	Nazwa dodatku musi być unikalna w obrębie danej wypożyczalni dla każdego języka – ta sama nazwa nie może występować u dwóch różnych dodatków w tym samym języku, ale identyczne nazewnictwo w innych językach jest dopuszczalne.
RB-OFE-13	Dezaktywacja dodatku może nastąpić tylko wtedy, gdy nie jest on przypisany do żadnej aktywnej oferty.
RB-OFE-14	Trwałe usunięcie dodatku jest dopuszczalne wyłącznie w sytuacji, gdy dany dodatek nie został przypisany ani do jakiejkolwiek aktywnej oferty, ani do żadnej rezerwacji.
RB-OFE-15	Zmniejszenie maksymalnej ilości danego dodatku jest niemożliwe w przypadku istnienia rezerwacji zamawiających większą liczbę tego dodatku.

5.3.4.4. Powiązania z innymi modułami

Moduł Oferty łączy moduł Flota z modułem Rezerwacje – definiuje warunki komercyjne na pojazdach zarządzanych w *5.3.2* i jest źródłem *snapshotów* parametrów dla rezerwacji *5.3.5*.

Tabela 24 Powiązania modułu Oferty z innymi modułami systemu. Źródło: opracowanie własne.

Moduł	Zakres powiązania
Flota	Oferta jest powiązana z konkretnym pojazdem lub grupą pojazdów; przypisanie do oddziału wynika z lokalizacji samego pojazdu albo z oddziału wskazanego przy tworzeniu oferty dla floty.
Rezerwacje	Rezerwacja jest tworzona wyłącznie na podstawie aktywnej oferty; parametry komercyjne (cena, limit przebiegu, depozyt oraz procent zwrotu) są kopiowane do rezerwacji jako niezmienny snapshot – nawet jeśli później wprowadzono zmiany w ofercie.
Strona publiczna wypożyczalni	Aktywne oferty są automatycznie publikowane na stronie internetowej dostępnej dla klientów wypożyczalni – to podstawowy kanał pozyskiwania nowych rezerwacji.

5.3.5. Rezerwacje

Moduł Rezerwacje stanowi kluczowy element systemu, wokół którego organizowane są przepływy związane z Flotą, Ofertami, Płatnościami oraz procesami Operacyjnymi i Komunikacyjnymi. Każda rezerwacja jest tworzona na podstawie aktywnej oferty pojazdu w imieniu konkretnego klienta wypożyczalni z określonym przedziałem czasowym – przy jej utworzeniu kopiowane są parametry oferty (cena, limit przebiegu, procent zwrotu depozytu oraz kwota i waluta) wraz z danymi osobowymi klienta jako niezależny *snapshot*. Cykl życia rezerwacji obejmuje siedem stanów: oczekująca, potwierdzona, aktywna i zakończona w ścieżce podstawowej oraz trzy zakończenia wyjątkowe – anulowana, nieodebrana lub przerwana (np. w przypadku konfliktu serwisowego). Status rezerwacji nie podlega bezpośredniej modyfikacji przez użytkownika; każda zmiana wynika z konkretnej akcji systemowej (utworzenie, potwierdzenie, wydanie pojazdu lub jego odbiór).

5.3.5.1. Aktorzy modułu

Tabela 25 Aktorzy modułu Rezerwacje. Źródło: opracowanie własne.

Aktor	Rola w module
Klient wypożyczalni	Samodzielnie tworzy rezerwacje na stronie publicznej wypożyczalni; dokonuje płatności; anuluje własne rezerwacje; przegląda ich historię.
Pracownik biurowy	Tworzy rezerwacje w imieniu klientów (również niezarejestrowanych w systemie); ręcznie potwierdza rezerwacje opłacane offline; edytuje rezerwacje przed wydaniem pojazdu; zmienia przypisanie pojazdu; anuluje rezerwacje.
Właściciel wypożyczalni	Wykonuje wszystkie operacje pracownika biurowego dla rezerwacji we wszystkich oddziałach wypożyczalni.
System automatyzacji	Anuluje rezerwacje po upływie czasu na potwierdzenie płatności; wysyła przypomnienia 24 godziny przed rozpoczęciem wynajmu; oznacza stanem „brak stawienia się" rezerwacje, których data zakończenia minęła bez wydania pojazdu.

5.3.5.2. Wymagania funkcjonalne

Tworzenie rezerwacji przez klienta:

- **WF-REZ-01:** Klient samodzielnie konfiguruje rezerwację na stronie publicznej wypożyczalni, wybierając aktywną ofertę, przedział czasowy mieszczący się w okresie obowiązywania oferty, listę dodatków z opcjonalnymi ilościami oraz metodę płatności online lub offline; system prezentuje wyliczoną cenę końcową oraz politykę zwrotów obowiązującą dla wybranej oferty.

- **WF-REZ-02:** Po zatwierdzeniu konfiguracji system tworzy rezerwację w stanie „oczekująca”, emituje powiadomienie e-mail z potwierdzeniem oraz wstępną wersją umowy najmu i – dla płatności online – kieruje klienta do bramki płatniczej.

Tworzenie rezerwacji przez pracownika biurowego:

- **WF-REZ-03:** Pracownik biurowy tworzy rezerwację w imieniu klienta, wybierając ofertę wynajmu pojazdu z floty dostępnej dla danego oddziału oraz określając szczegóły takie jak: termin rezerwacji (zakres dat), dodatkowe usługi, metodę płatności offline (gotówka albo przelew) i opcjonalne dane do faktury. Rezerwacja jest tworzona w stanie „oczekująca”.
- **WF-REZ-04:** Pracownik biurowy potwierdza rezerwację oczekującą, przenosząc ją do stanu „potwierdzona” – zazwyczaj po zaksięgowaniu wpłaty offline (gotówka lub przelew), ale również bez konieczności natychmiastowego rejestrowania płatności, np. w przypadku stałego klienta objętego programem zaufania i odroczonym terminem rozliczenia.

Potwierdzenie rezerwacji online:

- **WF-REZ-05:** Po otrzymaniu potwierdzenia autoryzacji płatności z bramki system automatycznie przenosi rezerwację ze stanu „oczekująca” do „potwierdzona”.

Edycja rezerwacji:

- **WF-REZ-06:** Pracownik biurowy edytuje rezerwację w stanach “oczekująca” albo “potwierdzona”; zakres edycji obejmuje termin wynajmu, listę dodatków, dzienny limit przebiegu, procent zwrotu, kwotę depozytu, dane do faktury oraz dane osobowe klienta.

Zmiana pojazdu w rezerwacji:

- **WF-REZ-07:** Pracownik biurowy zmienia pojazd w rezerwacji o statusie “oczekująca”, “potwierdzona” lub “przerwana”, wybierając aktywną ofertę innego pojazdu z tego samego oddziału, dostępnego przez cały okres trwania rezerwacji.
- **WF-REZ-08:** Po pomyślnej zmianie status rezerwacji przechodzi w “potwierdzona”; parametry komercyjne oferty (cena, limit przebiegu, procent zwrotu) pozostają niezmienione – zachowany jest ich stan z momentu pierwotnego utworzenia.

Anulowanie rezerwacji:

- **WF-REZ-09:** Klient anuluje własną rezerwację w stanach “oczekująca” albo “potwierdzona”; system rejestruje datę anulowania, oznacza ją jako wykonaną po stronie klienta i oblicza kwotę zwrotu zgodnie z procentem zwrotu zapisanym w rezerwacji.
- **WF-REZ-10:** Pracownik biurowy anuluje rezerwację w stanach “oczekująca” albo “potwierdzona”, podając przyczynę oraz stronę inicjującą (klient lub wypożyczalnia); jeśli anulowanie następuje po stronie wypożyczalni, klient otrzymuje pełny zwrot wpłaconej kwoty niezależnie od warunków rezerwacji.
- **WF-REZ-11:** Po anulowaniu rezerwacja przechodzi do stanu “anulowana”; powiązane z nią wykorzystanie pojazdu jest zwalniane, a proces zwrotu środków realizowany w module Płatności (*5.3.7*).

Automatyczne zarządzanie stanami:

- **WF-REZ-12:** System automatyzacji anuluje rezerwacje, dla których nie potwierdzono płatności w czasie określonym dla wybranej metody płatności.

- **WF-REZ-13:** System automatyzacji wysyła klientowi przypomnienie e-mail dwadzieścia cztery godziny przed planowanym rozpoczęciem rezerwacji „potwierdzonej”. W przypadku zmiany terminu rezerwacji flaga wysłanego przypomnienia jest zerowana.
- **WF-REZ-14:** System automatyzacji oznacza jako „nieodebrane” rezerwacje w stanie „oczekujące”, dla których data zakończenia rezerwacji powiększona o okres karencji oddziału minęła bez wydania pojazdu. Okres karencji, definiowany jest na poziomie oddziału (*5.3.1*).

Przerwanie rezerwacji wskutek konfliktu z serwisem pojazdu:

- **WF-REZ-15:** Utworzenie serwisu pojazdu w okresie obejmującym aktywną rezerwację tego pojazdu skutkuje przeniesieniem rezerwacji w stan „przerwana”. Wykorzystanie pojazdu w ramach tej rezerwacji jest skracane do dnia rozpoczęcia serwisu, a powiązane rozliczenie pozostaje rozliczane według zrealizowanej części najmu.

Wydanie i odbiór pojazdu (delegacja do 4.4.8):

- **WF-REZ-16:** Pracownik operacyjny podpisuje wygenerowaną z szablonu wypożyczalni umowę najmu lub protokół zdawczo-odbiorczy dla rezerwacji w stanach „potwierdzona” (umowa) albo „aktywna” (protokoły); szczegółowy przebieg wydania pojazdu oraz protokołu zdawczo-odbiorczego opisany jest w *5.3.8*.

5.3.5.3. Reguły biznesowe i walidacje

Tabela 26 Reguły biznesowe i walidacje modułu Rezerwacje. Źródło: opracowanie własne.

Nr	Reguła
RB-REZ-01	Każda rezerwacja jest powiązana dokładnie z jedną aktywną ofertą oraz z jednym aktywnym profilem klienta w momencie utworzenia.
RB-REZ-02	Okres trwania rezerwacji musi mieścić się w okresie obowiązywania powiązanej oferty i przypisania pojazdu do oddziału.
RB-REZ-03	Wykorzystanie tego samego pojazdu (rezerwacje, serwisy lub ręczne blokady) w ramach jednego przypisania do oddziału nie może się nakładać czasowo.
RB-REZ-04	Pojazd o stanie innym niż „aktywny” nie może zostać dodany do nowej rezerwacji.
RB-REZ-05	Klient posiadający dezaktywowany profil lub anonimowy nie może być powiązany z nową rezerwacją.
RB-REZ-06	Parametry komercyjne oferty (cena, limit przebiegu, procent zwrotu depozytu oraz kwota i waluta) wraz z danymi osobowymi klienta są kopiowane do rezerwacji jako snapshot – ich późniejsza edycja nie wpływa na dane zapisane przy tworzeniu.
RB-REZ-07	Rezerwacja utworzona przez klienta wymaga dokładnie jednej metody płatności online; rezerwacja stworzona przez pracownika biurowego musi mieć określoną metodę offline.
RB-REZ-08	Status „oczekująca” nie może zostać przywrócony po opuszczeniu tego stanu w cyklu życia rezerwacji.
RB-REZ-09	Przejście między statusami jest możliwe wyłącznie według ścisłej sekwencji: „oczekująca” -> „potwierdzona” -> „aktywna”; stan końcowy to „zakończona”.
RB-REZ-10	Anulować rezerwację można tylko ze statusów „oczekująca” lub „potwierdzona”, natomiast przejście do „przerwana” jest dopuszczalne wyłącznie z poziomu aktywnej.
RB-REZ-11	Edycja rezerwacji jest możliwa tylko w stanach początkowych: „oczekująca” lub „potwierdzona”. W pozostałych statusach system traktuje ją jako niezmienną (tylko do odczytu).
RB-REZ-12	Zmiana pojazdu w rezerwacji dopuszczalna jest pod warunkiem, że nowa oferta dotyczy tego samego oddziału i pozostaje dostępna przez cały okres trwania pierwotnej.
RB-REZ-13	Anulowanie rezerwacji z inicjatywy wypożyczalni uprawnia klienta do pełnego zwrotu wpłaconej kwoty; anulowanie przez użytkownika skutkuje zwrotem obliczonym według procentowego współczynnika zapisanego przy rezerwacji.
RB-REZ-14	Rezerwacja jest powiązana z jednym rozliczeniem sprzedażowym, który zawiera pozycję podstawową (odpowiadającą ofercie) oraz ewentualne dodatki; finalizacja tego dokumentu potwierdza rezerwację i uniemożliwia jej anulowanie.

5.3.5.4. Powiązania z innymi modułami

Moduł Rezerwacje jest kręgosłupem operacyjnym systemu – wokół rezerwacji organizowane są przepływy Floty, Ofert, Płatności, Operacyjnego i Komunikacji.

Tabela 27 Powiązania modułu Rezerwacje z innymi modułami systemu. Źródło: opracowanie własne.

Moduł	**Zakres powiązania**
Flota	Rezerwacja blokuje pojazd w wybranym okresie poprzez rezerwację; konflikt z serwisem przerywa proces tworzenia rezerwacji.
Klienci	Rezerwacja jest powiązana z klientem, a jego dane osobowe stanowią snapshot niezależny od późniejszych zmian w profilu użytkownika.
Oferty	Rezerwację można utworzyć wyłącznie na podstawie aktywnej oferty; parametry komercyjne (np. cena, warunki) są kopiowane z oferty do rezerwacji.
Finanse	Rozliczenie sprzedażowe generowane jest równolegle z rezerwacją. Jego finalizacja potwierdza status rezerwacji, natomiast anulowanie rozliczenia skutkuje automatycznym unieważnieniem rezerwacji.
Płatności	Potwierdzenie płatności online za rezerwację finalizuje proces; jeśli płatność nie zostanie potwierdzona w określonym czasie, system automatycznie anuluje rezerwację. Moduł Płatności obsługuje również zwroty środków.
Operacyjny	Proces wydania pojazdu (przejście rezerwacji do stanu „aktywna”) oraz odbioru (stan „zakończona”) jest realizowany przez moduł Operacyjny.
Komunikacja	Moduł Komunikacja wysyła powiadomienia e-mail: potwierdzenie utworzenia rezerwacji, przypomnienie 24 godziny przed rozpoczęciem wynajmu, informację o anulowaniu oraz status „nieodebrana”.

5.3.6. Finanse

Moduł Finanse jest odpowiedzialny za dokumenty rozliczeniowe prowadzone w systemie po obu stronach relacji platforma – wypożyczalnia – klient. Rozliczenie integruje sprzedawane lub kupowane elementy, takie jak rezerwacje z dodatkami czy subskrypcje pakietów, z aktualnym stanem zapłaty oraz procesem fakturowania. Moduł rozróżnia dwa kierunki rozliczeń: sprzedaż (gdy wypożyczalnia wystawia najem klientowi) i zakup (gdy wypożyczalnia nabywa subskrypcję pakietu od platformy). Obsługa transakcji, kanałów płatności online/offline oraz zwrotów realizowana jest przez osobny moduł Płatności (*5.3.7*), podczas gdy Finanse koncentrują się na opisie dokumentu rozliczeniowego i jego cyklu życia. Faktura VAT generowana jest wyłącznie w zewnętrznym systemie ERP, do którego rozliczenia są synchronizowane – Wellhicle nie nalicza podatku ani nie nadaje numeracji fakturom.

5.3.6.1. Aktorzy modułu

Tabela 28 Aktorzy modułu Finanse. Źródło: opracowanie własne.

Aktor	Rola w module
Właściciel wypożyczalni	Pełny wgląd w rozliczenia sprzedażowe wystawione dla swojej wypożyczalni oraz w rozliczenia zakupowe otrzymane od platformy; konfiguracja opcji rozliczeniowych wypożyczalni.
Pracownik biurowy	Edycja rozliczeń sprzedażowych w imieniu wypożyczalni: dane klienta, pozycje, anulowanie.
Administrator platformy	Zarządzanie rozliczeniami zakupowymi wystawianymi przez platformę za subskrypcje; centralna konfiguracja danych firmy do fakturowania i terminów płatności subskrypcji.
Klient wypożyczalni	Wgląd we własne dokumenty rozliczeniowe oraz pobieranie plików z nimi powiązanych.
System automatyzacji	Generowanie cyklicznych rozliczeń zakupowych za kolejne okresy subskrypcji; automatyczne anulowanie oczekujących rozliczeń po przekroczeniu terminu płatności.
Zewnętrzny system ERP*(aktor techniczny)*	Pobieranie dokumentów z platformy, wystawianie faktur VAT po swojej stronie oraz raportowanie postępu synchronizacji.

5.3.6.2. Wymagania funkcjonalne

Dokument rozliczeniowy:

- **WF-FIN-01:** Rozliczenia prowadzone są w dwóch kierunkach: sprzedaż (wypożyczalnia -> klient wypożyczalni, źródłem są rezerwacje i ich dodatki) oraz zakup (platforma -> wypożyczalnia, źródłem są subskrypcje pakietów).
- **WF-FIN-02:** Dla każdego rozliczenia przechowywane są: waluta, status, data wystawienia, data operacji, data sprzedaży, termin płatności oraz łączna kwota.
- **WF-FIN-03:** Rozliczenie sprzedażowe jest powiązane z oddziałem wypożyczalni, w którym powstała rezerwacja.
- **WF-FIN-04:** Na rozliczeniu zapisywany jest snapshot danych klienta oraz opcjonalnie snapshot danych firmy klienta – wartości te są kopiowane w momencie utworzenia dokumentu i nie aktualizują się automatycznie po zmianach w profilu klienta.
- **WF-FIN-05:** Rozliczenia są rozróżniane na fakturowe (gdy klient zaznaczył życzenie otrzymania faktury i dostarczył dane firmy) oraz bezfakturowe (paragon).
- **WF-FIN-06:** Wszystkie kwoty są reprezentowane w jednostkach minimalnych waluty jako liczby całkowite; ceny pozycji są kwotami brutto – system nie utrzymuje rozbicia na wartości netto, VAT i brutto.

Cykl życia rozliczenia:

- **WF-FIN-07:** Rozliczenie przechodzi przez statusy oczekujący -> opłacony -> zakończony, z gałęzią anulowany dostępną w każdym stanie edytowalnym.
- **WF-FIN-08:** Status opłacony jest ustawiany automatycznie po pokryciu łącznej kwoty dokumentu sumą wpłat.
- **WF-FIN-09:** Jeśli edycja lub zwrot obniży saldo poniżej całkowitej należności, status rozliczenia zmienia się z opłaconego na oczekujący; finalizacja powiązanej rezerwacji/subskrypcji nie jest cofana.

- **WF-FIN-10:** Właściciel wypożyczalni lub administrator platformy może ręcznie zakończyć rozliczenie oznaczone jako opłacone, jeżeli dalsza edycja nie jest już potrzebna.
- **WF-FIN-11:** Rozliczenia w statusie zakończony lub anulowany są tylko do odczytu – modyfikacja pozycji, kwot, snapshotu czy dodawanie płatności jest niedozwolone.
- **WF-FIN-12:** Snapshot klienta/firmy oraz pozycje rozliczenia można edytować wyłącznie w stanach oznaczonych jako edytowalne (oczekujący lub opłacony).

Pozycje rozliczeń:

- **WF-FIN-13:** Pozycje rozliczenia reprezentują dokładnie jedną z trzech rzeczy: rezerwację, dodatek do rezerwacji albo subskrypcję pakietu.
- **WF-FIN-14:** Dla każdej pozycji przechowywane są: nazwa, opcjonalny opis, cena jednostkowa, ilość, procent zwrotu (0-100) oraz wartości oryginalne (z momentu utworzenia) niezależnie od wartości bieżących (po korektach).
- **WF-FIN-15:** Łączna kwota rozliczeń jest sumą aktualnych wartości pozycji i przeliczana jest automatycznie po każdej zmianie pozycji.

Powstawanie rozliczeń:

- **WF-FIN-16:** Rozliczenie sprzedażowe jest generowane równocześnie z rezerwacją, gdzie pierwszą pozycją jest sama rezerwacja, a kolejne stanowią wybrane dodatki.
- **WF-FIN-17:** Rozliczenie zakupowe powstaje w momencie nabycia subskrypcji platformy, podczas aktualizacji planu w trakcie jej trwania oraz przy każdym kolejnym cyklu rozliczeniowym.
- **WF-FIN-18:** Dla planu darmowego rozliczenie nie jest tworzone – subskrypcja na tym poziomie pomija etap rozliczenia.
- **WF-FIN-19:** Przy generowaniu cyklicznego rozliczenia zakupowego daty wystawienia, operacji i sprzedaży są ustawiane na dzień bieżący, a termin płatności – na dzień bieżący powiększony o liczbę dni określoną w centralnej konfiguracji platformy.
- **WF-FIN-20:** Przed finalnym utworzeniem dokumentu dostępny jest podgląd rozliczenia – bez zapisu w systemie.

Saldo i płatności:

- **WF-FIN-21:** Saldo rozliczenia oblicza się jako sumę wpłat w kierunku podstawowym, pomniejszoną o zwroty; płatności anulowane oraz te z wykonanymi zwrotami nie są uwzględniane.
- **WF-FIN-22:** Dla każdego rozliczenia prezentuje się: kwotę łączną, zapłaconą i pozostałą do uregulowania.
- **WF-FIN-23:** Nadpłata jest blokowana – suma wpłat pierwotnych nie może przekroczyć całkowitej wartości dokumentu.
- **WF-FIN-24:** Obsługę płatności (kanały online i offline, statusy transakcji, terminy ważności oraz zwroty) realizuje moduł Płatności (*5.3.7* – delegacja).

Finalizacja zakupu:

- **WF-FIN-25:** Po pełnym opłaceniu rozliczenia sprzedażowego powiązana rezerwacja jest automatycznie zmieniana ze statusu oczekująca na potwierdzona.
- **WF-FIN-26:** Po pełnym opłaceniu rozliczenia zakupowego powiązana subskrypcja jest automatycznie aktywowana, jeżeli była w stanie zaplanowanym i data jej rozpoczęcia już minęła.

Zmiana planu subskrypcji:

- **WF-FIN-27:** Przy zmianie planu w trakcie subskrypcji system wylicza koszt jako sumę pozostałej części okresu według nowego planu pomniejszoną o niewykorzystany kredyt z bieżącego planu. Jeśli wynik jest dodatni, generowane jest rozliczenie zakupowe na kwotę różnicy; w przypadku wyniku zerowego lub ujemnego zmiana następuje bez tworzenia dokumentu rozliczeniowego.

Anulowanie i przeterminowanie:

- **WF-FIN-28:** Anulowanie rozliczenia wymaga niepustej przyczyny, oznaczenia strony inicjującej (klient lub wypożyczalnia) oraz czasu anulowania; wszystkie te dane są przechowywane na dokumencie.
- **WF-FIN-29:** Po anulowaniu rozliczenia sprzedażowego powiązana rezerwacja jest zmieniana w status anulowana, a powiązane użycie pojazdu jest dezaktywowane.
- **WF-FIN-30:** Po anulowaniu rozliczenia zakupowego powiązana subskrypcja zostaje wygaszona; jeżeli wygaszana subskrypcja była następną w łańcuchu przedłużeń, powiązanie u poprzedniej subskrypcji zostaje zerwane.
- **WF-FIN-31:** Po anulowaniu rozliczenia oczekujące płatności w kierunku podstawowym z zerową kwotą zapłaconą są anulowane.
- **WF-FIN-32:** System w tle automatycznie anuluje rozliczenia oczekujące, dla których minął termin płatności i nie zarejestrowano żadnej aktywnej płatności pierwotnej; anulowanie automatyczne uruchamia tę samą kaskadę co anulowanie ręczne (rezerwacja, subskrypcja, płatności).

Integracja z ERP i fakturowanie:

- **WF-FIN-33:** Po finalizacji rozliczenia otrzymuje status synchronizacji z systemem ERP równy do pobrania.
- **WF-FIN-34:** System przyjmuje od systemu ERP aktualizacje statusu synchronizacji: pobrany (ERP odebrał dokument), zafakturowany (ERP wystawił fakturę), błąd (ERP zwrócił błąd przetwarzania).
- **WF-FIN-35:** Dokumenty wystawione przez ERP (faktura) oraz dokumenty podsumowujące generowane wewnętrznie są załączane do rozliczenia i udostępniane uprawnionym aktorom.
- **WF-FIN-36:** System Wellhicle nie wystawia faktur VAT – naliczenie podatku, numeracja faktur i ewentualne korekty są realizowane wyłącznie po stronie systemu ERP.
- **WF-FIN-37:** Administrator platformy zarządza centralną konfiguracją rozliczeniową platformy: danymi firmy do wystawiania faktur za subskrypcje oraz domyślną liczbą dni terminu płatności dla rozliczeń zakupowych.

5.3.6.3. Reguły biznesowe i walidacje

Tabela 29 Reguły biznesowe i walidacje modułu Finanse. Źródło: opracowanie własne.

Nr	Reguła
RB-FIN-01	Rozliczenie zamknięte lub anulowane jest tylko do odczytu – system nie pozwala na modyfikację jakiegokolwiek pola, dodanie bądź usunięcie pozycji czy też wprowadzenie płatności.
RB-FIN-02	Status opłacony oznacza pełną zapłatę dokumentu; jednakże sam dokument pozostaje edytowalny – każda zmiana zwiększająca lub obniżająca saldo automatycznie aktualizuje status.
RB-FIN-03	Anulowanie rozliczenia jest jedyną metodą osiągnięcia stanu anulowany – bezpośrednia zmiana statusu na anulowany (bez procesu anulowania) jest zabroniona.
RB-FIN-04	Każde anulowanie rozliczenia wymaga podania niepustej przyczyny oraz informacji o stronie inicjującej tę operację.
RB-FIN-05	Anulowanie rozliczenia sprzedażowego powoduje automatyczne anulowanie powiązanej rezerwacji wraz z jej dodatkami – pozycje dotyczące dodatków nie podlegają osobnemu procesowi anulowania.
RB-FIN-06	Każde rozliczenie musi być przypisane dokładnie do jednej wypożyczalni i wystawiony w jednej walucie – brak możliwości mieszania walut w obrębie jednego dokumentu.
RB-FIN-07	Powiązanie z oddziałem jest wymagane wyłącznie dla rozliczeń sprzedażowych (dotyczących wynajmu pojazdów).
RB-FIN-08	Pozycja rozliczenia musi mieć dokładnie jedno powiązanie biznesowe – może to być rezerwacja, dodatek do rezerwacji lub subskrypcja; niedozwolone jest wskazywanie więcej niż jednego typu powiązania dla danej pozycji.
RB-FIN-09	Łączna kwota dokumentu zawsze równa się sumie aktualnych wartości jego pozycji – nie ma możliwości ręcznej modyfikacji tej wartości niezależnie od składowych.
RB-FIN-10	Procent zwrotu dla pozycji rozliczenia musi mieścić się w zakresie od 0% do 100%.
RB-FIN-11	Suma wpłat nie może przekroczyć całkowitej kwoty dokumentu – system blokuje nadpłaty.
RB-FIN-12	Żądanie faktury wymaga obecności snapshotu danych firmy powiązanej z rozliczeniem.
RB-FIN-13	Snapshot danych klienta i firmy zapisany na rozliczeniu jest niezmienny – nawet jeśli później nastąpią zmiany w profilu klienta, jedynym sposobem aktualizacji tych danych na dokumencie jest ponowna edycja rozliczenia w stanie edytowalnym.
RB-FIN-14	Plan darmowy nie generuje rozliczenia zakupowego (subskrypcyjnego).
RB-FIN-15	Podwyższenie subskrypcji przy różnicy zerowej lub ujemnej nie tworzy nowego rozliczenia ani nie uruchamia procedury zwrotu środków.
RB-FIN-16	Wszystkie kwoty prezentowane na dokumencie są wartościami brutto – system nie przechowuje oddzielnych pól dla kwot netto, VAT i brutto; operacje zawsze odbywają się w ujęciu brutto.

5.3.6.4. Powiązania z innymi modułami

Tabela 30 Powiązania modułu Finanse z innymi modułami systemu. Źródło: opracowanie własne.

Moduł	**Zakres powiązania**
Klienci	Snapshot danych klienta oraz jego firmy jest kopiowany do rozliczenia sprzedażowego w momencie utworzenia i nie aktualizuje się po późniejszych zmianach profilu.
Oferty	Ceny z ofert są uwzględniane w snapshocie rezerwacji, a następnie przenoszone do pozycji rozliczenia sprzedażowego.
Rezerwacje	Rozliczenie sprzedażowe jest generowane równocześnie z rezerwacją; jego finalizacja potwierdza rezerwację, natomiast anulowanie rozliczenia skutkuje odwołaniem rezerwacji i dezaktywacją powiązanego wykorzystania pojazdu.
Płatności	Każda płatność jest przypisana do konkretnego rozliczenia; saldo rozliczenia wynika ze stanu jego płatności, a zwroty są traktowane jako płatności w kierunku przeciwnym.
Pakiety	Subskrypcje na pakiety generują rozliczenia zakupowe – początkowo przy zakupie lub zmianie pakietu, a następnie cyklicznie zgodnie z harmonogramem określonym w konfiguracji pakietu.
Administracja platformy	Administrator platformy zarządza rozliczeniami zakupowymi za subskrypcje oraz centralną konfiguracją rozliczeniową (w tym danymi firmowymi platformy i domyślnym terminem płatności).
Komunikacja	Powiadomienia o zdarzeniach rozliczeniowych, takich jak zakończenie procesu rozliczenia, anulowanie czy dostępność faktury, są wysyłane przez moduł Komunikacja.

5.3.7. Płatności

Moduł Płatności obsługuje pojedyncze transakcje finansowe powiązane z rozliczeniami modułu Finanse w obu kierunkach rozliczenia: sprzedaży (klient opłaca rezerwację wraz z dodatkami) oraz zakupu (wypożyczalnia opłaca platformie subskrypcję wybranego pakietu). Każda płatność jest osadzona w jednym rozliczeniu, ma określoną metodę realizacji (online lub offline), kierunek przepływu środków, planowaną i zrealizowaną kwotę w walucie rozliczenia oraz status; zwrot jest reprezentowany jako odrębna płatność o kierunku przeciwnym, powiązana z płatnością źródłową. Moduł zapewnia obsługę dwóch kanałów: płatności online za pośrednictwem zewnętrznej bramki oraz płatności offline (gotówka, przelew bankowy lub inne metody manualne), a także zarządzanie katalogiem metod płatności – osobno dla każdej wypożyczalni oraz centralnie dla platformy.

5.3.7.1. Aktorzy modułu

Tabela 31 Aktorzy modułu Płatności. Źródło: opracowanie własne.

Aktor	Rola w module
Klient wypożyczalni	Inicjuje płatność za rezerwację podczas jej tworzenia online – przez bramkę płatniczą (online) lub realizując wpłatę offline (gotówka, przelew bankowy) zgodnie z instrukcją określoną dla wybranej metody.
Pracownik biurowy	Rejestruje płatności offline za rezerwacje po otrzymaniu wpłaty od klienta; inicjuje zwroty środków oraz aktualizuje kwoty i statusy płatności sprzedażowych.
Właściciel wypożyczalni	Wykonuje wszystkie operacje pracownika biurowego po stronie sprzedażowej, konfiguruje katalog metod płatności swojej wypożyczalni oraz inicjuje płatność za subskrypcję platformy.
Administrator platformy	Rejestruje płatności offline za subskrypcje platformy, inicjuje zwroty środków w obszarze zakupowym oraz konfiguruje centralny katalog metod płatności używany dla rozliczeń.
Bramka płatnicza	Realizuje płatności online inicjowane zarówno przez klientów (kierunek sprzedażowy), jak i przez wypożyczalnie (kierunek zakupowy).
System automatyzacji	Anuluje płatności oraz powiązane rezerwacje lub subskrypcje po upływie czasu na realizację płatności określonego dla wybranej metody.

5.3.7.2. Wymagania funkcjonalne

Zarządzanie metodami płatności:

- **WF-PLA-01:** Właściciel wypożyczalni ma możliwość zarządzania katalogiem metod płatności – dodaje, edytuje i trwale usuwa metody offline (gotówka, przelew bankowy lub inna forma manualnej zapłaty) oraz online za pośrednictwem bramki płatniczej. Dla każdej zdefiniowanej metody określa czas automatycznego anulowania nieopłaconych płatności, status aktywności (czy jest dostępna) oraz widoczność dla klientów podczas procesu rezerwacji online. Trwałe usunięcie metody płatności możliwe jest tylko wtedy, gdy nie istnieją powiązane z nią transakcje.
- **WF-PLA-02:** Administrator platformy ma uprawnienia do centralnego zarządzania katalogiem metod płatności używanych wyłącznie przy rozliczaniu subskrypcji za korzystanie z systemu (pakiety SaaS). Metody te nie są dostępne dla wypożyczalni podczas obsługi rezerwacji pojazdów – analogicznie metody płatności dodane przez poszczególne wypożyczalnie pozostają niedostępne przy rozliczeniach subskrypcji platformy.

Płatność sprzedażowa (klient wypożyczalni -> wypożyczalnia):

- **WF-PLA-03:** Klient wypożyczalni, tworząc rezerwację online, wybiera widoczną dla niego metodę płatności z katalogu dostępnego w ramach oferty wypożyczalni – online lub offline – a system generuje pojedynczą transakcję pokrywającą całość rozliczenia.
- **WF-PLA-04:** W przypadku wyboru metody online klient jest kierowany do bramki płatniczej; system oznacza status transakcji jako zrealizowaną (przy poprawnym zakończeniu transakcji) lub anulowaną (w przypadku niepowodzenia, wyświetlając komunikat o możliwości ponowienia płatności).
- **WF-PLA-05:** Dla metody offline klient otrzymuje instrukcję realizacji wpłaty (np. dane do przelewu lub lokalizację wypożyczalni) zdefiniowaną na poziomie wybranej metody płatności; transakcja pozostaje w statusie „oczekująca” do momentu ręcznego potwierdzenia wpłaty przez pracownika biurowego lub automatycznego anulowania po upływie określonego czasu.
- **WF-PLA-06:** Pracownik biurowy rejestruje płatność offline podając zapłaconą kwotę; po pełnym rozliczeniu status płatności zmienia się na „zrealizowana”.

- **WF-PLA-07:** Pracownik biurowy może modyfikować wcześniej zarejestrowaną transakcję offline (korekta kwoty, aktualizacja daty wygaśnięcia lub anulowanie nieopłaconej płatności).

Płatność zakupowa (wypożyczalnia -> platforma):

- **WF-PLA-08:** Właściciel wypożyczalni przy zakupie nowej subskrypcji pakietu, podczas podniesienia planu w trakcie aktywnej subskrypcji wybiera widoczną dla swojej wypożyczalni metodę płatności z katalogu platformy.
- **WF-PLA-09:** W przypadku wyboru płatności online za subskrypcję, po autoryzacji w bramce system oznacza transakcję jako zrealizowaną (moduł Finanse potwierdza opłacenie rozliczenia zakupowego i aktywuje subskrypcję) lub anulowaną (przy niepowodzeniu autoryzacji – otrzymuje komunikat z możliwością ponowienia próby płatności).
- **WF-PLA-10:** Administrator platformy rejestruje wpłatę offline za subskrypcję, podając kwotę już rozliczoną; po zaksięgowaniu pełnej wymaganej sumy rozliczenia płatność zmienia status na „zrealizowana".
- **WF-PLA-11:** Administrator platformy może zaktualizować już zarejestrowaną wpłatę offline (korekta kwoty rozliczonej, ustawienie nowej daty wygaśnięcia płatności lub anulowanie nieopłaconej transakcji).

Zwrot środków:

- **WF-PLA-12:** Po stronie sprzedażowej pracownik biurowy inicjuje zwrot środków powiązany z konkretną, zrealizowaną płatnością klienta – podając kwotę oraz metodę płatności do wykorzystania. Po stronie zakupowej analogiczny proces uruchamia administrator platformy. Zwrot jest rejestrowany jako nowa płatność o przeciwnym kierunku w stosunku do transakcji źródłowej i pozostaje z nią powiązany.
- **WF-PLA-13:** Po ręcznym wprowadzeniu zwrotu, osoba inicjująca oznacza kwotę jako wypłaconą (gotówka wydana klientowi lub środki przelane).
- **WF-PLA-14:** Dla płatności realizowanych przez bramkę system automatycznie uruchamia transakcję zwrotną i aktualizuje status operacji po otrzymaniu potwierdzenia z zewnętrznego dostawcy usług płatniczych.

Obsługa cyklicznych zdarzeń:

- **WF-PLA-15:** System automatyzacji po upływie czasu wyznaczonego przez metodę płatności anuluje płatność, dla której nie zarejestrowano żadnej kwoty zrealizowanej. Po stronie sprzedażowej następuje automatyczne anulowanie powiązanej rezerwacji. Po stronie zakupowej – automatyczne anulowanie rozliczenia oraz wygaśnięcie subskrypcji.

5.3.7.3. Reguły biznesowe i walidacje

Tabela 32 Reguły biznesowe i walidacje modułu Płatności. Źródło: opracowanie własne.

Nr	Reguła
RB-PLA-01	Każda płatność jest powiązana dokładnie z jednym rozliczeniem oraz z dokładnie jedną metodą płatności; metoda należy do katalogu wypożyczalni (dla rozliczenia sprzedażowego) albo do centralnego katalogu platformy (dla rozliczenia zakupowego).
RB-PLA-02	Kierunek płatności pierwotnej jest wyznaczany przez typ rozliczenia: dla rozliczenia sprzedażowego – uznanie, a dla rozliczenia zakupowego – obciążenie. Zwrot ma kierunek przeciwny do płatności źródłowej.
RB-PLA-03	Płatność może mieć jeden z czterech statusów: „oczekująca", „zrealizowana", „anulowana" lub „zwrócona". Status nie może powrócić do stanu „oczekująca" po jego opuszczeniu.
RB-PLA-04	Płatność można anulować wyłącznie wtedy, gdy zarejestrowana kwota zrealizowana wynosi zero.
RB-PLA-05	Płatność źródłową można oznaczyć jako zwróconą tylko wtedy, gdy zarejestrowana kwota zrealizowana jest dodatnia.
RB-PLA-06	Zwrot środków reprezentowany jest jako odrębna płatność o kierunku przeciwnym do płatności źródłowej, powiązana z nią jednoznaczną referencją.
RB-PLA-07	Kwota zwrotu nie może przekraczać kwoty zrealizowanej płatności źródłowej.
RB-PLA-08	Płatności i zwroty nie mogą być modyfikowane ani inicjowane, jeśli powiązane rozliczenie jest w stanie zakończonym lub anulowanym.
RB-PLA-09	Płatności nie podlegają trwałemu usunięciu – każda transakcja pozostaje zapisana w historii powiązanej z rozliczeniem.
RB-PLA-10	Konfiguracja katalogu metod płatności wypożyczalni jest wyłącznym uprawnieniem właściciela wypożyczalni, a konfiguracja centralnego katalogu platformy – administratora platformy.
RB-PLA-11	Rejestrowanie i aktualizowanie płatności po stronie sprzedażowej (rozliczenie SALE) należy do właściciela wypożyczalni oraz pracownika biurowego, natomiast po stronie zakupowej (rozliczenie PURCHASE) – wyłącznie do administratora platformy.
RB-PLA-12	Trwałe usunięcie metody płatności jest możliwe tylko wtedy, gdy nie istnieją żadne płatności powiązane z tą metodą.
RB-PLA-13	Walutą płatności jest waluta powiązanego rozliczenia; waluty tej nie można zmienić w trakcie życia płatności.
RB-PLA-14	Metoda płatności niewidoczna dla klienta nie może zostać wybrana przy tworzeniu rezerwacji online – jest dostępna wyłącznie pracownikom biurowym podczas ręcznej rejestracji płatności.
RB-PLA-15	Wypożyczalnia umożliwia klientom tworzenie rezerwacji online tylko wtedy, gdy posiada co najmniej jedną aktywną i widoczną dla klienta metodę płatności.
RB-PLA-16	Platforma umożliwia zakup subskrypcji płatnego pakietu tylko, gdy dysponuje co najmniej jedną aktywną i widoczną dla wypożyczalni metodą płatności w katalogu platformy.

5.3.7.4. Powiązania z innymi modułami

Moduł Płatności obsługuje strumień pieniężny rozliczeń modułu Finanse w obu kierunkach: potwierdza status rezerwacji modułu Rezerwacje (sprzedaż) oraz aktywuje subskrypcję modułu Pakiety (zakup).

Tabela 33 Powiązania modułu Płatności z innymi modułami systemu. Źródło: opracowanie własne.

Moduł	Zakres powiązania
Rezerwacje	Potwierdzenie płatności online za rezerwację przez bramkę płatniczą wyzwala automatyczne potwierdzenie powiązanej rezerwacji; anulowanie płatności po upływie czasu prowadzi do automatycznego anulowania rezerwacji.
Finanse	Każda płatność jest pozycją finansową rozliczenia wystawionego w module Finanse; suma zrealizowanych płatności w kierunku podstawowym (pomniejszona o zwroty) wyznacza saldo rozliczenia i jego status.
Pakiety	Potwierdzenie płatności zakupowej za subskrypcję pakietu finalizuje proces rozliczenia zakupowego, co aktywuje subskrypcję; anulowanie płatności zakupowej po upływie czasu prowadzi do anulowania rozliczenia zakupowego i wygaszenia powiązanej subskrypcji.
Administracja platformy	Administrator platformy zarządza centralnym katalogiem metod płatności oraz rejestruje płatności po stronie zakupowej w imieniu platformy.
Komunikacja	Powiadomienia e-mail o wynikach transakcji (potwierdzenie wpłaty, informacja o zwrocie, anulowanie z powodu braku płatności) są wysyłane przez moduł Komunikacja po obu stronach rozliczenia.

5.3.8. Operacyjny

Moduł Operacyjny obsługuje fizyczną realizację najmu w terenie – przekazanie pojazdu klientowi z potwierdzonej rezerwacji, jego odbiór po zakończeniu wynajmu oraz zgłaszanie usterek wykrytych podczas eksploatacji. Procedury wydania i odbioru są prowadzone przez pracowników operacyjnych za pomocą aplikacji mobilnej i składają się z wieloetapowych procesów: weryfikacja klienta, inspekcja stanu technicznego pojazdu (przebieg, paliwo, opony, karoseria), dokumentacja fotograficzna oraz podpisanie protokołu przez klienta. Każdy etap zmienia status procedury (oczekujące -> rozpoczęte -> przygotowane -> zakończone). Zamknięcie procesu wydania lub odbioru automatycznie aktualizuje status rezerwacji oraz wyliczany stan pojazdu. Moduł pełni rolę łącznika między częścią administracyjną systemu a działaniami fizycznymi w oddziale.

5.3.8.1. Aktorzy modułu

Tabela 34 Aktorzy modułu Operacyjny. Źródło: opracowanie własne.

Aktor	Rola w module
Pracownik operacyjny	Prowadzi procedurę wydania pojazdu klientowi (weryfikacja, inspekcja, podpisanie umowy) oraz procedurę odbioru pojazdu po zwrocie (inspekcja, podpisanie protokołu); zgłasza usterki wykryte przy odbiorze albo w trakcie eksploatacji.
Właściciel wypożyczalni	Wykonuje wszystkie operacje pracownika operacyjnego oraz ma podgląd wszystkich operacji wykonanych w obrębie wypożyczalni.
Klient wypożyczalni	Składa podpis elektroniczny na umowie najmu przy odbiorze pojazdu oraz na protokole zdawczo-odbiorczym przy zwrocie pojazdu.

5.3.8.2. Wymagania funkcjonalne

Podpis pracownika operacyjnego:

- **WF-OPE-01:** Pracownik operacyjny przy pierwszym logowaniu w aplikacji mobilnej przesyła swój podpis odręczny; może go zaktualizować w dowolnym momencie.

Wydanie pojazdu:

- **WF-OPE-02:** Aplikacja mobilna prezentuje pracownikowi operacyjnemu listę planowanych wydań pojazdów, opartą na potwierdzonych rezerwacjach w jego oddziale.
- **WF-OPE-03:** Pracownik operacyjny rozpoczyna procedurę wydania pojazdu, weryfikując dane dokumentu tożsamości klienta: numer dokumentu i kraj wydania.
- **WF-OPE-04:** Pracownik operacyjny rejestruje wynik inspekcji pojazdu przed wydaniem, uwzględniając aktualny stan licznika, poziom paliwa, stan ogumienia i karoserii, opcjonalny komentarz oraz dokumentację fotograficzną.
- **WF-OPE-05:** Klient składa podpis elektroniczny na umowie najmu wygenerowanej z szablonu wypożyczalni; po podpisaniu system zapisuje finalną wersję dokumentu.
- **WF-OPE-06:** Po podpisaniu umowy procedura wydania jest kończona – rezerwacja oznaczana jako aktywna, a wykorzystanie pojazdu jako rozpoczęte; finalna wersja umowy najmu zostaje przesłana klientowi e-mailem.

Odbiór pojazdu:

- **WF-OPE-07:** Aplikacja mobilna prezentuje pracownikowi operacyjnemu listę planowanych odbiorów opartą na aktywnych rezerwacjach w jego oddziale.
- **WF-OPE-08:** Pracownik operacyjny rejestruje wynik inspekcji pojazdu po zwrocie: stan licznika, poziom paliwa, stan ogumienia i karoserii, opcjonalny komentarz oraz dokumentację fotograficzną.
- **WF-OPE-09:** Klient składa podpis elektroniczny na protokole zdawczo-odbiorczym wygenerowanym z szablonu wypożyczalni; po podpisaniu system zapisuje finalną wersję protokołu.
- **WF-OPE-10:** Po podpisaniu protokołu procedura odbioru jest kończona – system oznacza rezerwację jako zakończoną i zamyka wykorzystanie pojazdu; finalny protokół jest wysyłany klientowi e-mailem.

Zgłaszanie usterek:

- **WF-OPE-11:** Pracownik operacyjny może zgłosić usterkę pojazdu w dowolnym momencie eksploatacji (najczęściej podczas odbioru pojazdu po zwrocie), wprowadzając dane dotyczące: identyfikatora pojazdu, szczegółowego opisu usterki oraz – jeśli to konieczne – załączając zdjęcia dokumentujące problem. Dodatkowo istnieje możliwość powiązania zgłoszenia z rezerwacją, w trakcie której usterka wystąpiła. Takie zgłoszenie tworzy serwis pojazdu w module Flota.

5.3.8.3. Reguły biznesowe i walidacje

Tabela 35 Reguły biznesowe i walidacje modułu Operacyjny. Źródło: opracowanie własne.

Nr	Reguła
RB-OPE-01	Procedura wydania pojazdu jest dostępna wyłącznie dla rezerwacji w stanie „potwierdzona” (potwierdzonej).
RB-OPE-02	Procedura odbioru pojazdu jest dostępna wyłącznie dla rezerwacji w stanie „aktywna”.
RB-OPE-03	Procedura wydania pojazdu prowadzona jest w kontrolowanej sekwencji kroków: weryfikacja klienta -> inspekcja stanu technicznego -> podpis klienta na umowie najmu; cofanie kroków nie jest możliwe.
RB-OPE-04	Procedura odbioru pojazdu prowadzona jest w kontrolowanej sekwencji kroków: inspekcja stanu technicznego -> podpis klienta na protokole zdawczo-odbiorczym; cofanie kroków nie jest możliwe.
RB-OPE-05	Zakończenie procedury wydania pojazdu wymaga uprzedniego przeprowadzenia inspekcji stanu technicznego oraz złożenia podpisu przez klienta na umowie najmu.

Nr	Reguła
RB-OPE-06	Zakończenie procedury odbioru pojazdu wymaga uprzedniego przeprowadzenia inspekcji stanu technicznego oraz złożenia podpisu przez klienta na protokole zdawczo-odbiorczym.
RB-OPE-07	Zakończenie procedury wydania pojazdu powoduje automatyczne przejście rezerwacji w stan „aktywna”; od tego momentu rezerwacja jest dostępna wyłącznie do odczytu (bez możliwości edycji lub anulowania).
RB-OPE-08	Zakończenie procedury odbioru pojazdu powoduje automatyczne przejście rezerwacji w stan „zakończona”.
RB-OPE-09	Procedurę wydania i odbioru pojazdu może realizować wyłącznie pracownik operacyjny przypisany do oddziału, na którym znajduje się dany pojazd, lub właściciel wypożyczalni.
RB-OPE-10	Każdy pracownik operacyjny przed rozpoczęciem procedur wydania/odbioru musi posiadać zarejestrowany podpis w systemie.
RB-OPE-11	Inspekcja stanu technicznego pojazdu obejmuje rejestrację: licznika kilometrów, poziomu paliwa, ogumienia, karoserii oraz opcjonalnie zdjęć i komentarza; każda procedura wydania/odbioru generuje osobny wpis stanu technicznego dla danego pojazdu.
RB-OPE-12	Zgłoszenie usterki podczas odbioru skutkuje automatycznym utworzeniem serwisu powiązanego z rezerwacją, w której wykryto problem (moduł Flota).
RB-OPE-13	Aktywna rezerwacja objęta nowym zleceniem serwisowym obejmującym okres wynajmu jest automatycznie przerywana ze statusem „przerwana”.

5.3.8.4. Powiązania z innymi modułami

Moduł Operacyjny realizuje fizyczne kroki cyklu życia rezerwacji i pojazdu – operuje na encjach modułów Flota, Rezerwacje, Komunikacja oraz Zarządzanie wypożyczalnią (szablony dokumentów).

Tabela 36 Powiązania modułu Operacyjny z innymi modułami systemu. Źródło: opracowanie własne.

Moduł	Zakres powiązania
Zarządzanie wypożyczalnią	Szablony umowy najmu i protokołu zdawczo-odbiorczego konfigurowane w module Zarządzanie wypożyczalnią są wykorzystywane do generowania finalnych dokumentów podczas procedur wydania oraz odbioru pojazdu.
Flota	Procedura wydania i odbioru pojazdu rejestruje jego stan techniczny (przebieg, paliwo, zużycie opon, karoserię) oraz wykonane zdjęcia w module Flota. Zgłoszenie usterki skutkuje utworzeniem zlecenia serwisowego opisanego również w tym module.
Rezerwacje	Zakończenie procedury wydania pojazdu zmienia status rezerwacji na „aktywna”, natomiast finalizacja odbioru przenosi ją do stanu „zakończona”. Obie aktualizacje wynikają bezpośrednio z realizacji procesów w module Rezerwacje.
Komunikacja	Po sfinalizowaniu procedury wydania klient wypożyczalni otrzymuje elektroniczną kopię umowy najmu, a po odbiorze – protokół zdawczo-odbiorczy w formie załącznika e-mail.

5.3.9. Strona publiczna wypożyczalni

Moduł Strona publiczna wypożyczalni stanowi fasadę systemu dla klientów wypożyczalni – gości przeglądających ofertę oraz zarejestrowanych użytkowników składających rezerwacje. Prezentuje aktywne oferty, dane kontaktowe wypożyczalni i jej oddziałów, umożliwia rejestrację/logowanie oraz składanie rezerwacji. Po zalogowaniu klient uzyskuje dostęp do osobistego panelu, gdzie może przeglądać historię rezerwacji, dokumenty oraz edytować swoje dane. Moduł ten nie wprowadza nowych encji ani samodzielnych funkcjonalności – agreguje i udostępnia klientom wypożyczalni dane z modułów nadrzędnych w odpowiednio przystosowanej formie.

5.3.9.1. Aktorzy modułu

Tabela 37 Aktorzy modułu Strona publiczna wypożyczalni. Źródło: opracowanie własne.

Aktor	Rola w module
Gość wypożyczalni	Niezarejestrowany użytkownik strony; przegląda aktywne oferty, dane wypożyczalni i jej oddziałów; może przejść do rejestracji konta lub logowania.
Klient wypożyczalni	Zalogowany użytkownik z aktywnym kontem; ma dostęp do wszystkich funkcji gościa oraz do procesu składania rezerwacji i osobistego panelu klienta.

5.3.9.2. Wymagania funkcjonalne

Prezentacja katalogu ofert:

- **WF-SWY-01:** Strona publiczna prezentuje listę aktywnych ofert wypożyczalni; dla każdej oferty wyświetla dane pojazdu (model, rok produkcji, typ nadwozia, rodzaj paliwa i skrzynię biegów), informacje o oddziale oraz szczegóły cenowe: dzienną stawkę najmu wraz z walutą depozytu zwrotnego. Dodatkowo pokazuje limit przebiegu na dobę oraz listę dostępnych dodatków (z cenami i sposobem rozliczenia).
- **WF-SWY-02:** Zarówno gość, jak i klient mogą filtrować i sortować oferty według oddziału, parametrów pojazdu (np. marka, rocznik), przedziału cenowego oraz okresu dostępności w kalendarzu rezerwacyjnym.
- **WF-SWY-03:** Po wybraniu konkretnej oferty użytkownik przechodzi do widoku szczegółów, gdzie znajduje pełny opis warunków najmu oraz interaktywny kalendarz z dostępnością pojazdu w wybranym terminie.

Prezentacja danych wypożyczalni:

- **WF-SWY-04:** Strona publiczna prezentuje dane kontaktowe wypożyczalni (adres e-mail, numer telefonu) oraz listę aktywnych oddziałów z ich danymi kontaktowymi.
- **WF-SWY-05:** Strona publiczna udostępnia dokumenty regulaminowe wypożyczalni – politykę prywatności oraz regulamin serwisu – w formie plików do pobrania, dostępnych z każdej podstrony.

Rejestracja i logowanie klienta:

- **WF-SWY-06:** Gość przechodzi do formularza rejestracji konta klienta wypożyczalni; po wypełnieniu danych proces jest obsługiwany przez moduł Klienci, łącznie z weryfikacją adresu e-mail.
- **WF-SWY-07:** Klient loguje się do strony publicznej za pomocą adresu e-mail i hasła; opcjonalnie korzysta z procedury odzyskania hasła.

Składanie rezerwacji:

- **WF-SWY-08:** Zalogowany klient wypożyczalni, posiadający aktywny profil użytkownika, rozpoczyna proces składania rezerwacji z poziomu oferty; konfiguruje parametry rezerwacji (w tym wybór dodatków), określa metodę płatności oraz finalizuje transakcję.
- Panel klienta:
- **WF-SWY-09:** Zalogowany klient przegląda w panelu historię swoich rezerwacji, ich aktualne statusy, dane pojazdów i oddziałów oraz pobiera dokumenty rezerwacyjne (potwierdzenia, umowy najmu, protokoły zdawczo-odbiorcze, dokumenty rozliczeniowe).

- **WF-SWY-10:** Zalogowany klient edytuje w panelu swoje dane osobowe i kontaktowe, dane dokumentu tożsamości oraz dane firmy używane do faktur; zmienia również hasło do swojego konta.

5.3.9.3. Reguły biznesowe i walidacje

Tabela 38 Reguły biznesowe i walidacje modułu Strona publiczna wypożyczalni. Źródło: opracowanie własne.

Nr	Reguła
RB-SWY-01	Strona publiczna wypożyczalni jest dostępna wyłącznie dla aktywnych wypożyczalni; dezaktywacja uniemożliwia klientom wypożyczalni korzystanie z niej.
RB-SWY-02	Publikacja strony publicznej jest możliwa tylko po dodaniu aktualnych wersji dokumentów: polityki prywatności i regulaminu serwisu.
RB-SWY-03	Na stronie publicznej wyświetlane są jedynie aktywne oferty.
RB-SWY-04	Lista oddziałów prezentuje tylko te, które są aktualnie aktywne.
RB-SWY-05	Rezerwacja może zostać złożona tylko przez klienta, który jest zalogowany.
RB-SWY-06	Dostęp do panelu klienta dostępny jest po zalogowaniu się; użytkownik widzi tam wyłącznie rezerwacje, w których sam jest stroną transakcji.
RB-SWY-07	Złożenie rezerwacji online jest możliwe tylko wtedy, gdy dana wypożyczalnia udostępnia przynajmniej jedną aktywną i widoczną dla klienta metodę płatności.

5.3.9.4. Powiązania z innymi modułami

Moduł Strona publiczna wypożyczalni nie wprowadza nowych encji – agreguje dane z modułów subordynowanych.

Tabela 39 Powiązania modułu Strona publiczna wypożyczalni z innymi modułami systemu. Źródło: opracowanie własne.

Moduł	Zakres powiązania
Zarządzanie wypożyczalnią	Konfiguracja wyglądu strony, treści sekcji w obsługiwanych językach, dokumenty regulaminowe oraz dane kontaktowe wypożyczalni i jej oddziałów.
Klienci	Rejestracja, logowanie, odzyskiwanie hasła oraz edycja danych klienta przez panel klienta.
Oferty	Katalog ofert prezentowanych klientom wypożyczalni to filtrowany widok aktywnych ofert.
Rezerwacje	Proces składania rezerwacji (konfiguracja, walidacja, utworzenie) oraz historia rezerwacji klienta.
Płatności	Realizacja płatności za rezerwację w trakcie składania zamówienia.
Komunikacja	Wszystkie wiadomości e-mail kierowane do klienta wypożyczalni (potwierdzenia rejestracji, weryfikacja adresu, reset hasła, potwierdzenia rezerwacji, dokumenty) są wysyłane przez moduł Komunikacja.

5.3.10. Pakiety

Moduł Pakiety odpowiada za katalog pakietów subskrypcyjnych platformy oraz za zarządzanie subskrypcjami wypożyczalni – przypisywanie pakietów do konkretnych instancji, określanie ich obowiązywania, harmonogramu rozliczeń i automatycznego odnawiania. Pakiet definiuje limity ilościowe zasobów wypożyczalni (oddziały, pojazdy, pracownicy biurowi oraz operacyjni), domyślny okres trwania subskrypcji i harmonogram cyklicznych rozliczeń. Subskrypcja stanowi podstawę aktywności wypożyczalni w systemie – jej wygaśnięcie bez odnowienia skutkuje automatyczną dezaktywacją instancji. Katalog pakietów jest zarządzany centralnie przez administratora platformy,

natomiast właściciel może w trakcie życia subskrypcji wybrać lub zmienić pakiet, skonfigurować automatyczne odnawianie albo ręcznie przedłużyć okres obowiązywania.

5.3.10.1. Aktorzy modułu

Tabela 40 Aktorzy modułu Pakiety. Źródło: opracowanie własne.

Aktor	Rola w module
Administrator platformy	Zarządza katalogiem pakietów wypożyczalni (tworzy, edytuje, aktywuje, dezaktywuje) oraz może bezpośrednio przypisać wypożyczalni subskrypcję dowolnego pakietu (w tym pakietu nieaktywnego, np. testowego).
Właściciel wypożyczalni	Wybiera pakiet wypożyczalni podczas pierwszej konfiguracji, zmienia go w trakcie życia instancji, włącza i wyłącza automatyczne odnawianie subskrypcji oraz ręcznie ją przedłuża.
System automatyzacji	Cyklicznie odnawia subskrypcje z aktywną flagą automatycznego odnawiania, generuje rozliczenia w zaplanowanych terminach, aktualizuje statusy subskrypcji oraz dezaktywuje wypożyczalnie, których subskrypcja wygasła bez przedłużenia.

5.3.10.2. Wymagania funkcjonalne

Katalog pakietów:

- **WF-PAK-01:** Administrator platformy tworzy w katalogu pakiet, podając unikalny kod identyfikujący pakiet, nazwę handlową i opis w obsługiwanych językach platformy, status aktywności, limity (maksymalna liczba oddziałów, pojazdów, pracowników biurowych i operacyjnych – z możliwością ustawienia braku limitu w polu dowolnym), domyślny okres trwania subskrypcji wybrany z listy dozwolonych długości (jeden tydzień, dwa tygodnie, miesiąc, sześć miesięcy, rok), procent zwrotu przy anulowaniu, harmonogram cyklicznych rozliczeń (jednostka cyklu i krotność) oraz cennik w jednej lub wielu walutach.
- **WF-PAK-02:** Administrator zarządza istniejącym pakietem w katalogu – edytuje jego nazwę, opis i cennik oraz aktywuje albo dezaktywuje pakiet.

Zarządzanie subskrypcją wypożyczalni:

- **WF-PAK-03:** Właściciel wypożyczalni podczas pierwszej konfiguracji wybiera pakiet z listy aktywnych pakietów dostępnych w katalogu, określa walutę rozliczeń oraz opcjonalną datę rozpoczęcia subskrypcji i metodę płatności. System tworzy subskrypcję dla wypożyczalni na podstawie wybranego pakietu, stosując cennik obowiązujący w momencie utworzenia subskrypcji.
- **WF-PAK-04:** Administrator platformy może przypisać wypożyczalni dowolny pakiet z katalogu (w tym nieaktywny, np. testowy) – taka administracyjna konfiguracja jest niedostępna dla samodzielnego wyboru przez właściciela.
- **WF-PAK-05:** Właściciel może zmienić bieżący pakiet subskrypcji; system oblicza proporcjonalną różnicę kosztów za pozostałe dni w dotychczasowej subskrypcji, generuje odpowiednią korektę rozliczenia i aktualizuje limity oraz funkcje zgodnie z nowym pakietem.
- **WF-PAK-06:** Właściciel decyduje o włączeniu lub wyłączeniu automatycznego odnawiania bieżącej subskrypcji.

Cykliczne odnawianie i rozliczenia:

- **WF-PAK-07:** System automatycznego przedłużania subskrypcji wznawia ten sam pakiet na dzień jej wygaśnięcia, jeśli opcja auto odnowienia jest aktywna i pakiet pozostaje dostępny w katalogu; nowy okres rozliczeniowy zaczyna się następnego dnia według aktualnych stawek.

- **WF-PAK-08:** System automatycznie generuje cykliczne faktury dla aktywnych subskrypcji, stosując ustalony harmonogram (okres + częstotliwość + maksymalna liczba okresów).
- **WF-PAK-09:** System aktualizuje statusy subskrypcji: zmienia zaplanowaną na aktywną w dniu rozpoczęcia oraz oznacza nieodnowioną jako wygasłą po zakończeniu okresu ważności.

Dezaktywacja wypożyczalni:

- **WF-PAK-10:** System automatyzacji dezaktywuje wypożyczalnię, której subskrypcja wygasła i nie została odnowiona po upływie okresu karencji określonego w konfiguracji platformy.

5.3.10.3. Reguły biznesowe i walidacje

Tabela 41 Reguły biznesowe i walidacje modułu Pakiety. Źródło: opracowanie własne.

Nr	Reguła
RB-PAK-01	Kod identyfikujący pakiet jest unikalny w skali platformy.
RB-PAK-02	Każda wypożyczalnia posiada w danym momencie dokładnie jedną aktywną subskrypcję; dopuszczalna jest jedna zaplanowana na okres po zakończeniu obecnej.
RB-PAK-03	Limity zasobów (oddziały, pojazdy, pracownicy biurowi i operacyjni) wynikające z aktywnego pakietu są egzekwowane przy tworzeniu nowych zasobów.
RB-PAK-04	Dozwolony okres trwania subskrypcji wybierany jest z listy dopuszczalnych długości (tydzień, dwa tygodnie, miesiąc, sześć miesięcy lub rok). Dowolny niestandardowy okres nie jest dozwolony.
RB-PAK-05	Zmiana cennika pakietu nie wpływa na cenę aktywnej subskrypcji ani zaplanowanej; nowy cennik obowiązuje dopiero przy następnej utworzonej z tego pakietu.
RB-PAK-06	Edycja limitów pakietu jest niedopuszczalna, jeśli ma on już aktywne subskrypcje; w takim przypadku limity i okres trwania pozostają niezmienne.
RB-PAK-07	Dezaktywacja pakietu w katalogu nie kończy istniejących subskrypcji opartych na nim, ale uniemożliwia jego wybór do nowych lub odnowień.
RB-PAK-08	Automatyczne odnowienie subskrypcji wymaga, aby pakiet docelowy pozostawał aktywny w katalogu w momencie planowanego odnowienia.
RB-PAK-09	Zmiana pakietu dla wypożyczalni jest możliwa tylko wtedy, gdy bieżąca liczba zasobów mieści się w limitach nowego pakietu.
RB-PAK-10	Subskrypcja oparta na darmowym pakiecie (zerowe ceny lub brak pozycji cennika) nie generuje rozliczenia.
RB-PAK-11	Wypożyczalnia, której subskrypcja wygasła i nie została odnowiona, jest automatycznie dezaktywowana po upływie okresu karencji.
RB-PAK-12	Reaktywacja zdezaktywowanej wypożyczalni wymaga utworzenia nowej aktywnej subskrypcji opartej na pakiecie z katalogu.

5.3.10.4. Powiązania z innymi modułami

Moduł Pakiety stanowi fundament aktywności wypożyczalni i ramą limitów wszystkich modułów subordynowanych.

Tabela 42 Powiązania modułu Pakiety z innymi modułami systemu. Źródło: opracowanie własne.

Moduł	Zakres powiązania
Zarządzanie wypożyczalnią	Aktywna subskrypcja wypożyczalni jest niezbędnym warunkiem aktywności instancji; limity z pakietu są egzekwowane przy tworzeniu oddziałów i pracowników; przywracanie zdezaktywowanej wypożyczalni wymaga aktywnej subskrypcji.
Flota	Limit pojazdów z aktywnego pakietu jest sprawdzany przy tworzeniu pojazdu w module Flota.

Moduł	Zakres powiązania
Administracja platformy	Administrator platformy zarządza katalogiem pakietów oraz może bezpośrednio przypisywać subskrypcje wypożyczalniom poza standardową ścieżką zakupu.
Finanse	Zakupy i odnowienia subskrypcji generowane przez ten moduł stanowią dokumenty rozliczeniowe modułu Finanse; harmonogram cykli rozliczeniowych jest definiowany na poziomie pakietu.

5.3.11. Administracja platformy

Moduł Administracja platformy odpowiada za utrzymanie i nadzór całej platformy Wellhicle, w tym instancji wypożyczalni, konfiguracji globalnej oraz słowników danych referencyjnych. Administrator platformy zarządza katalogiem pakietów, katalogiem modeli pojazdów, czarną listą subdomen, globalnymi szablonami wiadomości i dokumentów oraz danymi rozliczeniowymi platformy. Może również podejmować bezpośrednie działania na dowolnej wypożyczalni, takie jak aktywacja, dezaktywacja, trwałe usunięcie, zmiana subdomeny czy przypisanie subskrypcji oraz zalogować się do niej z rolą właściciela.

5.3.11.1. Aktorzy modułu

Tabela 43 Aktorzy modułu Administracja platformy. Źródło: opracowanie własne.

Aktor	Rola w module
Administrator platformy	Jedyny aktor modułu; zarządza instancjami wypożyczalni, czarną listą subdomen, katalogiem modeli pojazdów, globalnymi szablonami dokumentów oraz danymi rozliczeniowymi platformy.

5.3.11.2. Wymagania funkcjonalne

Zarządzanie kontami administratorów:

- **WF-ADM-01:** Administrator platformy tworzy konto kolejnego administratora platformy, podając jego dane identyfikacyjne i adres e-mail.

Zarządzanie instancjami wypożyczalni:

- **WF-ADM-02:** Administrator platformy przegląda listę wszystkich instancji wypożyczalni, uwzględniając ich status, dane identyfikacyjne (nazwa handlowa, subdomena, informacje o właścicielu oraz aktywny pakiet), datę utworzenia i statystyki wykorzystania zasobów: liczbę oddziałów, pojazdów, pracowników, klientów wypożyczalni oraz aktywnych rezerwacji.
- **WF-ADM-03:** Administrator platformy ręcznie dezaktywuje wybraną wypożyczalnię – niezależnie od statusu jej subskrypcji, np. z powodu naruszenia regulaminu korzystania z systemu Wellhicle.
- **WF-ADM-04:** Administrator platformy ręcznie przywraca aktywność wcześniej dezaktywowanej wypożyczalni.
- **WF-ADM-05:** Administrator trwale usuwa zdezaktywowaną wypożyczalnię, likwidując powiązane dane: oddziały, pojazdy, użytkowników (pracownicy biurowi i operacyjni), rezerwacje oraz zapisy rozliczeniowe.
- **WF-ADM-06:** Administrator zmienia subdomenę wypożyczalni.
- **WF-ADM-07:** Administrator platformy loguje się do wybranej wypożyczalni z rolą jej właściciela (np. celem rozwiązania problemu technicznego).

Zarządzanie czarną listą subdomen:

- WF-ADM-08: Administrator platformy zarządza czarną listą subdomen zastrzeżonych przez system – przegląda ją, dodaje nowe wpisy i usuwa istniejące.

Zarządzanie słownikami danych referencyjnych:

- **WF-ADM-09:** Administrator platformy zarządza katalogiem modeli pojazdów – dodaje pojedyncze modele, importuje partię modeli z zewnętrznego źródła referencyjnego oraz trwale usuwa nieużywane modele.
- **WF-ADM-10:** Administrator platformy zarządza katalogiem globalnych szablonów dokumentów wykorzystywanych we wszystkich wypożyczalniach (umowa najmu, dokument rozliczenia, protokół zdawczo-odbiorczy oraz pozostałe predefiniowane typy) – wgrywa nowe wersje szablonów i podmienia istniejące; globalne szablony są dostarczane jako wzorce dla nowych wypożyczalni przy pierwszej konfiguracji oraz pozostają dostępne do skopiowania przez cały okres funkcjonowania instancji.

Konfiguracja rozliczeń platformy:

- **WF-ADM-11:** Administrator platformy konfiguruje dane firmowe wykorzystywane do wystawiania faktur subskrypcyjnych dla wypożyczalni.
- **WF-ADM-12:** Administrator ustawia domyślny termin płatności (w dniach) obowiązujący dla wszystkich rozliczeń subskrypcyjnych.

5.3.11.3. Reguły biznesowe i walidacje

Tabela 44 Reguły biznesowe i walidacje modułu Administracja platformy. Źródło: opracowanie własne.

Nr	Reguła
RB-ADM-01	Konto administratora platformy posiada uprawnienia ponad granicami instancji wypożyczalni; jego operacje nie są ograniczone do konkretnej wypożyczalni.
RB-ADM-02	Adres e-mail administratora platformy jest unikalny w skali całej platformy.
RB-ADM-03	Każda aktywna instancja wypożyczalni jest powiązana z dokładnie jednym kontem właściciela.
RB-ADM-04	Trwałe usunięcie instancji wypożyczalni jest możliwe wyłącznie po jej wcześniejszej dezaktywacji.
RB-ADM-05	Trwałe usunięcie instancji wypożyczalni jest operacją nieodwracalną – kasuje wszystkie dane powiązane z tą wypożyczalnią (oddziały, pojazdy, użytkownicy, rezerwacje, rozliczenia, płatności).
RB-ADM-06	Dezaktywacja instancji wypożyczalni przez administratora platformy unieważnia aktywne sesje wszystkich jej pracowników i klientów oraz blokuje im logowanie, niezależnie od statusu subskrypcji wypożyczalni.
RB-ADM-07	Subdomena każdej aktywnej wypożyczalni jest unikalna w skali całej platformy.
RB-ADM-08	System nie pozwala przypisać ani zmienić subdomeny wypożyczalni na nazwę znajdującą się na czarnej liście subdomen.
RB-ADM-09	Konfiguracja firmowych danych platformy oraz domyślnego terminu płatności subskrypcji wpływa wyłącznie na rozliczenia subskrypcyjne wystawiane od momentu zmiany; rozliczenia historyczne pozostają niezmienione.

5.3.11.4. Powiązania z innymi modułami

Moduł Administracja platformy jest nadrzędny wobec wszystkich modułów funkcjonalnych – administrator platformy operuje na encjach pozostałych modułów ponad granicami instancji wypożyczalni.

Tabela 45 Powiązania modułu Administracja platformy z innymi modułami systemu. Źródło: opracowanie własne.

Moduł	Zakres powiązania
Zarządzanie wypożyczalnią	Administrator platformy dezaktywuje i aktywuje dowolną instancję wypożyczalni oraz zmienia jej subdomenę; dezaktywacja przez administratora ma identyczny skutek jak dezaktywacja przez właściciela (RB-ZWY-15).
Pakiety	Zarządzanie katalogiem pakietów i bezpośrednie przypisywanie subskrypcji wypożyczalniom (w tym pakietów nieaktywnych).
Finanse	Konfiguracja danych firmowych platformy i domyślnego terminu płatności jest używana przy generowaniu rozliczeń subskrypcyjnych w module Finanse; szablon dokumentu rozliczeniowego (zarządzany w WF-ADM-10) jest globalnym wzorcem dostarczanym wypożyczalniom.
Płatności	Administrator platformy zarządza centralnym katalogiem metod płatności platformy oraz rejestruje i koryguje płatności po stronie zakupowej w imieniu platformy.
Operacyjny	Szablony umowy najmu i protokołu zdawczo-odbiorczego (zarządzane w WF-ADM-10) są globalnymi wzorcami dostarczanymi wypożyczalniom przy pierwszej konfiguracji i wykorzystywanymi przez moduł Operacyjny przy wydawaniu i odbiorze pojazdu.
Strona publiczna platformy	Konfiguracja katalogu pakietów oraz dane informacyjne platformy zarządzane przez administratora są prezentowane na stronie publicznej platformy.

5.3.12. Komunikacja

Moduł Komunikacja odpowiada za zautomatyzowaną komunikację systemu z klientami i właścicielami wypożyczalni poprzez wiadomości e-mail. System utrzymuje zamknięty katalog typów wiadomości, obsługujących kluczowe zdarzenia w pozostałych modułach: weryfikację adresu e-mail, reset hasła, powitanie klienta, potwierdzenie i przypomnienie rezerwacji, anulowanie rezerwacji, potwierdzenie płatności, fakturę, zmianę warunków planu subskrypcyjnego oraz test konfiguracji SMTP. Dla każdego typu wiadomości istnieje globalny szablon zarządzany przez administratora platformy – dodatkowo może być przypisana lokalna kopia szablonu, dostosowana do konkretnej wypożyczalni. Wypożyczalnia ma możliwość modyfikacji tematu i treści powiadomień dla swoich klientów lub całkowitej dezaktywacji wybranych typów komunikatów. Proces wysyłki realizowany jest z wykorzystaniem skonfigurowanego konta pocztowego danej wypożyczalni lub – w przypadku jego braku – stosowane jest domyślne konto platformy Wellhicle.

5.3.12.1. Aktorzy modułu

Tabela 46 Aktorzy modułu Komunikacja. Źródło: opracowanie własne.

Aktor	Rola w module
Administrator platformy	Zarządza globalnymi szablonami wiadomości e-mail platformy; ma podgląd lokalnych szablonów wszystkich wypożyczalni.
Właściciel wypożyczalni	Konfiguruje konto pocztowe nadawcze wypożyczalni; edytuje treść i temat lokalnych szablonów wiadomości oraz aktywuje albo dezaktywuje wysyłkę poszczególnych powiadomień.

5.3.12.2. Wymagania funkcjonalne

Zarządzanie globalnymi szablonami wiadomości:

- **WF-KOM-01:** Administrator platformy zarządza globalnymi szablonami wiadomości dla typów dopuszczających modyfikację (np. powitanie klienta, potwierdzenie rezerwacji) – edytuje ich treść i temat oraz aktywuje lub dezaktywuje szablon; zmiany wprowadzone w globalnym szablonie obowiązują te wypożyczalnie, które nie posiadają lokalnej kopii danego typu komunikatu. Dezaktywacja blokuje natomiast możliwość wysyłania takich wiadomości we wspomnianych wypożyczalniach.

Zarządzanie lokalnymi szablonami wiadomości wypożyczalni:

- **WF-KOM-02:** Właściciel wypożyczalni zarządza lokalnymi szablonami wiadomości: tworzy lokalną kopię globalnego szablonu dla wybranego typu (lokalny szablon ma pierwszeństwo przed globalnym w obrębie tej wypożyczalni), edytuje treść i temat tego szablonu, aktywuje lub dezaktywuje go (dezaktywacja blokuje możliwość wysyłania takich wiadomości z poziomu danej wypożyczalni) oraz trwale usuwa lokalny szablon, co powoduje przywrócenie użycia globalnego szablonu dla tego typu.

Konto pocztowe wypożyczalni:

- **WF-KOM-03:** Właściciel konfiguruje nadawcze konto pocztowe wypożyczalni, podając adres e-mail, hasło oraz parametry serwera SMTP (port i protokół szyfrowania). Po skonfigurowaniu konta inicjuje testową wiadomość na swój prywatny adres, aby zweryfikować poprawność ustawień.
- **WF-KOM-04:** Właściciel trwale usuwa skonfigurowane konto pocztowe wypożyczalni – od tego momentu komunikacja e-mail jest realizowana wyłącznie przez systemowe konto domenowe platformy Wellhicle.

Generowanie i dostarczanie wiadomości:

- **WF-KOM-05:** Wystąpienie zdarzenia w pozostałych modułach systemu (utworzenie konta klienta, weryfikacja adresu e-mail, reset hasła, utworzenie i anulowanie rezerwacji, przypomnienie dwadzieścia cztery godziny przed rozpoczęciem rezerwacji, potwierdzenie płatności, wystawienie dokumentu rozliczeniowego, zmiana warunków planu subskrypcyjnego) skutkuje wygenerowaniem wiadomości e-mail o określonym typie. System automatycznie dobiera obowiązujący szablon dla danej wypożyczalni (lokalny lub globalny), wybiera konto pocztowe nadawcze (skonfigurowane indywidualnie dla wypożyczalni lub systemowe konto domenowe platformy) i dostarcza wiadomość do adresata.

5.3.12.3. Reguły biznesowe i walidacje

Tabela 47 Reguły biznesowe i walidacje modułu Komunikacja. Źródło: opracowanie własne.

Nr	Reguła
RB-KOM-01	Katalog predefiniowanych typów wiadomości e-mail jest zarządzany przez system; użytkownicy wypożyczalni ani administrator platformy nie mogą dodawać nowych typów lub usuwać istniejących.
RB-KOM-02	Dla każdego typu wiadomości dostępny jest dokładnie jeden globalny szablon zarządzany przez administratora platformy; wypożyczalnia może utworzyć maksymalnie jedną lokalną kopię takiego szablonu dla danego typu.
RB-KOM-03	Lokalny szablon wiadomości stosowany przez wypożyczalnię nadpisuje globalny podczas generowania komunikatów; aktualizacja szablonu globalnego nie wpływa na istniejące lokalne kopie.
RB-KOM-04	Wysłanie wiadomości danego typu wymaga aktywnego szablonu (lokalnego dla danej wypożyczalni lub globalnego, jeśli brak lokalnej kopii).
RB-KOM-05	Treść i temat szablonów mogą zawierać znaczniki zmiennych. Aktualizacja niedozwolona w przypadku nieobsługiwanych dla danego typu tagów.
RB-KOM-06	Użytkownicy systemu nie mają możliwości ręcznego inicjowania wysyłki e-maili – komunikaty są generowane wyłącznie automatycznie jako reakcja na zdarzenia systemowe.
RB-KOM-07	Konto pocztowe przypisane do wypożyczalni konfiguruje jej właściciel; pracownicy biurowi i operacyjni nie posiadają uprawnień do edycji tych ustawień.
RB-KOM-08	W przypadku błędu podczas próby wysyłki przez konto wypożyczalni, system automatycznie wykorzystuje awaryjne konto domenowe platformy do dostarczenia komunikatu.

5.3.12.4. Powiązania z innymi modułami

Moduł Komunikacja jest pomostem między pozostałymi modułami systemu a kanałem e-mailowym – konsumuje zdarzenia z modułów subordynowanych i wysyła wiadomości w odpowiedzi.

Tabela 48 Powiązania modułu Komunikacja z innymi modułami systemu. Źródło: opracowanie własne.

Moduł	Zakres powiązania
Zarządzanie wypożyczalnią	Konfiguracja konta pocztowego nadawczego wypożyczalni jest wykonywana w obrębie tego modułu; status aktywności wypożyczalni warunkuje możliwość wysyłki wiadomości w jej imieniu.
Klienci	Wiadomości typu „weryfikacja adresu", „reset hasła", „powitanie klienta" są generowane w odpowiedzi na zdarzenia rejestracji i resetu hasła obsługiwane w module Klienci.
Rezerwacje	Wiadomości potwierdzające utworzenie, anulowanie i zbliżający się termin rezerwacji są generowane w odpowiedzi na zdarzenia w module Rezerwacje.
Płatności	Wiadomości potwierdzające zrealizowane wpłaty i zwroty są generowane w odpowiedzi na zdarzenia w module Płatności.

Moduł	Zakres powiązania
Pakiety	Wiadomości informujące właściciela o zmianie warunków planu subskrypcyjnego są generowane w odpowiedzi na zdarzenia w module Pakiety.
Finanse	Wiadomości z załączonymi dokumentami rozliczeniowymi są generowane w odpowiedzi na ich wystawienie w module Finanse.

5.3.13. Strona publiczna platformy

Moduł Strona publiczna platformy stanowi ogólnodostępny interfejs systemu Wellhicle, nieprzypisany do konkretnej wypożyczalni. Pełni funkcję informacyjno-promocyjną – prezentuje opis możliwości platformy oraz katalog dostępnych pakietów subskrypcyjnych i służy jako punkt startowy dla nowych właścicieli, którzy zakładają konto. Po zalogowaniu użytkownicy są przekierowywani do kreatora pierwszej konfiguracji swojej instancji wypożyczalni. Moduł nie definiuje odrębnych encji domenowych, lecz agreguje dane z modułów podrzędnych, a proces tworzenia nowej wypożyczalni jest delegowany do odpowiedniego komponentu systemowego.

5.3.13.1. Aktorzy modułu

Tabela 49 Aktorzy modułu Strona publiczna platformy. Źródło: opracowanie własne.

Aktor	Rola w module
Gość platformy SaaS	Niezarejestrowany odwiedzający stronę publiczną platformy; przegląda informacje o platformie i katalog pakietów; inicjuje rejestrację konta właściciela wypożyczalni.

5.3.13.2. Wymagania funkcjonalne

Prezentacja platformy:

- **WF-SPL-01:** Strona publiczna platformy prezentuje opis systemu Wellhicle, jego funkcjonalności oraz korzyści dla wypożyczalni pojazdów, a także dane kontaktowe operatora platformy.
- **WF-SPL-02:** Na stronie publicznej wyświetlany jest katalog aktywnych pakietów subskrypcyjnych przeznaczonych do samodzielnego wyboru przez właścicieli wypożyczalni, zawierający informacje o limitach zasobów, domyślnym okresie trwania oraz cenniku w obsługiwanych walutach.

Rejestracja właściciela wypożyczalni:

- **WF-SPL-03:** Gość platformy zakłada konto właściciela wypożyczalni, podając adres e-mail, hasło, imię i nazwisko oraz opcjonalnie numer telefonu; po zatwierdzeniu formularza system automatycznie wysyła wiadomość weryfikacyjną na podany adres.
- **WF-SPL-04:** Po otwarciu linku zawartego w tej wiadomości konto zostaje aktywowane; przy pierwszym logowaniu użytkownik jest automatycznie kierowany do procesu tworzenia nowej wypożyczalni za pomocą dedykowanego kreatora konfiguracji.

5.3.13.3. Reguły biznesowe i walidacje

Tabela 50 Reguły biznesowe i walidacje modułu Strona publiczna platformy. Źródło: opracowanie własne.

Nr	Reguła
RB-SPL-01	Strona publiczna platformy prezentuje wyłącznie aktywne pakiety subskrypcyjne; dezaktywowane opcje nie są widoczne dla użytkowników.

Nr	Reguła
RB-SPL-02	Rejestracja konta właściciela wymaga akceptacji regulaminu oraz polityki prywatności platformy Wellhicle.
RB-SPL-03	Adres e-mail przypisany do właściciela musi być unikalny – próba rejestracji z już istniejącym adresem (użytym przez innego właściciela lub administratora) kończy się odrzuceniem.
RB-SPL-04	Założenie konta właściciela nie tworzy automatycznie wypożyczalni; instancja wypożyczalni powstaje dopiero po ukończeniu konfiguracji początkowej.
RB-SPL-05	Aktywacja konta właściciela następuje poprzez kliknięcie linku weryfikacyjnego przesłanego e-mailem po rejestracji.

5.3.13.4. Powiązania z innymi modułami

Moduł Strona publiczna platformy nie wprowadza własnych encji – agreguje dane modułów platformowych i deleguje proces tworzenia wypożyczalni.

Tabela 51 Powiązania modułu Strona publiczna platformy z innymi modułami systemu. Źródło: opracowanie własne.

Moduł	Zakres powiązania
Zarządzanie wypożyczalnią	Po pierwszym logowaniu właściciela kreator pierwszej konfiguracji.
Pakiety	Katalog aktywnych pakietów prezentowanych na stronie platformy pochodzi z modułu Pakiety.
Administracja platformy	Konfiguracją katalogu pakietów oraz danymi informacyjnymi platformy zarządza administrator za pośrednictwem modułu Administracja platformy.
Komunikacja	Wiadomość weryfikacyjna wysyłana po rejestracji właściciela oraz powiadomienie o aktywacji konta są obsługiwane przez moduł Komunikacja.

5.4. Wymagania niefunkcjonalne

W niniejszym podrozdziale przedstawiono wymagania niefunkcjonalne systemu Wellhicle. W przeciwieństwie do wymagań funkcjonalnych, które zostały opisane odrębnie dla każdego modułu w podrozdziale *5.3* wyżej, wymagania niefunkcjonalne zestawiono w jednym miejscu, ponieważ dotyczą one głównie systemu jako całości, a nie poszczególnych modułów.

Wymagania niefunkcjonalne uzupełniają opis systemu o warunki dotyczące sposobu jego działania, określając cechy, które powinien zachowywać niezależnie od wykonywanej operacji. Na potrzeby pracy wymagania pogrupowano w siedem obszarów tematycznych odpowiadających specyfice projektu.

5.4.1. Bezpieczeństwo i ochrona danych

Wymagania w tym obszarze koncentrują się na bezpieczeństwie dostępu do systemu, ochronie danych poszczególnych wypożyczalni oraz zabezpieczeniu komunikacji między aplikacjami a serwerem. Kluczowe znaczenie ma izolacja danych różnych wypożyczalni, ponieważ system funkcjonuje jako platforma obsługująca wiele niezależnych podmiotów.

Obszar ten obejmuje przede wszystkim mechanizmy uwierzytelniania i autoryzacji użytkowników, ochronę sesji, szyfrowanie komunikacji oraz bezpieczne przechowywanie danych dostępowych.

- **NF-BZ-01:** Dostęp do funkcji systemu wykraczających poza publicznie dostępne treści wymaga uwierzytelnienia użytkownika. System stosuje tokenowy mechanizm uwierzytelniania z ograniczonym czasem ważności tokenu dostępowego (60 minut) oraz mechanizmami jego odświeżania i unieważniania.
- **NF-BZ-02:** Każda operacja udostępniana przez interfejs programistyczny systemu podlega autoryzacji opartej na rolach aktorów, określonych w podrozdziale *5.3*; użytkownik może wykonać jedynie operacje przypisane do jego roli.
- **NF-BZ-03:** Dane poszczególnych wypożyczalni są odizolowane: operacje wykonane na danych jednej wypożyczalni nie mogą ujawniać ani modyfikować informacji dotyczących innych wypożyczalni.
- **NF-BZ-04:** System nie przechowuje haseł użytkowników w postaci jawnej ani możliwej do odczytania. W bazie danych zapisywana jest wyłącznie zabezpieczona reprezentacja hasła.

5.4.2. Wydajność i skalowalność

Wymagania w tym obszarze koncentrują się na efektywnym działaniu systemu przy rosnącym obciążeniu, minimalizując ryzyko przeciążeń spowodowanych pojedynczymi operacjami lub wypożyczalniami. Uwzględniono również możliwość dalszej skalowalności systemu wraz ze zwiększaniem liczby obsługiwanych wypożyczalni i użytkowników.

- **NF-WY-01:** Wszystkie listy danych zwracane przez interfejs API są stronicowane; domyślny rozmiar strony wynosi 20 elementów, a maksymalny to 100 elementów.
- **NF-WY-02:** Wysyłanie wiadomości e-mail odbywa się w tle i nie wstrzymuje odpowiedzi systemu po wykonaniu operacji.
- **NF-WY-03:** Strona internetowa wypożyczalni ładuje się w mniej niż sekundę (przy łączu o prędkości pobierania min. 100Mbps).
- **NF-WY-04:** Rozmiar pojedynczego pliku załącznika przesyłanego do systemu jest ograniczony do 10 MB.

5.4.3. Niezawodność

Wymagania dotyczące tego obszaru koncentrują się na ciągłości działania systemu oraz odporności na błędy. Ich głównym celem jest minimalizacja ryzyka wystąpienia sytuacji, w których awaria pojedynczego elementu mogłaby skutkować utratą danych lub trwałą niedostępnością platformy.

- **NF-NZ-01:** System umożliwia wykrywanie niesprawnych komponentów serwerowych. W przypadku wykrycia problemu dany komponent może zostać automatycznie zrestartowany w celu przywrócenia jego działania.
- **NF-NZ-02:** W przypadku niepowodzenia wysyłki wiadomości e-mail system ponawia próbę jej wysłania. Nieudane wysyłki są rejestrowane, aby możliwa była późniejsza analiza błędów.
- **NF-NZ-03:** Operacje obejmujące kilka powiązanych zmian w danych są wykonywane jako jedna całość. Oznacza to, że albo wszystkie zmiany zostają zapisane, albo w przypadku błędu system wycofuje całą operację
- **NF-NZ-04:** Pliki użytkowników są przechowywane w magazynie obiektowym niezależnym od cyklu życia aplikacji.

5.4.4. Użyteczność interfejsów

Wymagania z tego obszaru dotyczą wygody i czytelności korzystania z systemu w aplikacjach przeglądarkowych oraz aplikacji mobilnej. Ich znaczenie wynika z tego, że z platformy korzystają różne grupy użytkowników, wykonujące odmienne zadania.

- **NF-UX-01:** Publiczna strona wypożyczalni jest responsywna i dostosowuje układ interfejsu do ekranów komputerów oraz urządzeń mobilnych.
- **NF-UX-02:** Formularze walidują dane wejściowe i prezentują użytkownikowi czytelne komunikaty o błędach, wskazujące konkretne pole oraz przyczynę odrzucenia wprowadzonych informacji.
- **NF-UX-03:** Procesy wydania i zwrotu pojazdu są dostępne w dedykowanej aplikacji mobilnej dla pracowników operacyjnych.
- **NF-UX-04:** System umożliwia korzystanie z interfejsu w jasnym oraz ciemnym motywie kolorystycznym.

5.4.5. Wielojęzyczność i lokalizacja

Wymagania w tym obszarze koncentrują się na dostosowaniu systemu do obsługi wypożyczalni działających w różnych językach, walutach i strefach czasowych. Ich celem jest zapewnienie spójnego przetwarzania danych specyficznych dla każdej konkretnej wypożyczalni, uwzględniając jej indywidualne ustawienia.

- **NF-LO-01:** Interfejs panelu do zarządzania wypożyczalnią oraz aplikacja mobilna są dostępne w języku polskim i angielskim.
- **NF-LO-02:** Publiczna strona wypożyczalni prezentuje treści w językach skonfigurowanych przez właściciela wypożyczalni. Dzięki temu zakres wersji językowych strony może być dostosowany do potrzeb konkretnej wypożyczalni.
- **NF-LO-03:** Publiczna strona wypożyczalni umożliwia prezentowanie ofert w wielu walutach skonfigurowanych przez właściciela wypożyczalni.
- **NF-LO-04:** System wspiera ustawienia regionalne (ang. *locale*), dzięki czemu sposób prezentacji dat, godzin i walut może być dostosowany do języka wybranego dla danej wypożyczalni**.**

5.4.6. Utrzymanie i monitorowanie systemu

Wymagania z tego obszaru dotyczą utrzymania systemu po wdrożeniu oraz możliwości kontrolowania jego działania. Ich celem jest ułatwienie diagnozowania błędów, testowania zmian oraz rozwijania platformy.

- **NF-UM-01:** System zapewnia mechanizmy monitorowania działania platformy, obejmujące centralne gromadzenie logów, śledzenie żądań przechodzących przez komponenty systemu oraz prezentowanie kluczowych metryk technicznych.
- **NF-UM-02:** Interfejs programistyczny systemu jest udokumentowany w formacie OpenAPI, a dokumentacja generowana automatycznie na podstawie kodu, co gwarantuje jej aktualność i zgodność ze stanem faktycznym API.
- **NF-UM-03:** Proces budowania, testowania i wdrażania komponentów systemu jest zautomatyzowany; wdrożenie nowej wersji nie wymaga ręcznej konfiguracji środowiska.

5.4.7. Kompatybilność i przenośność

Wymagania w tym obszarze dotyczą możliwości uruchamiania systemu w różnych środowiskach oraz jego współpracy z zewnętrznymi systemami, uwzględniając zarówno aplikacje użytkownika, jak i komponenty serwerowe platformy.

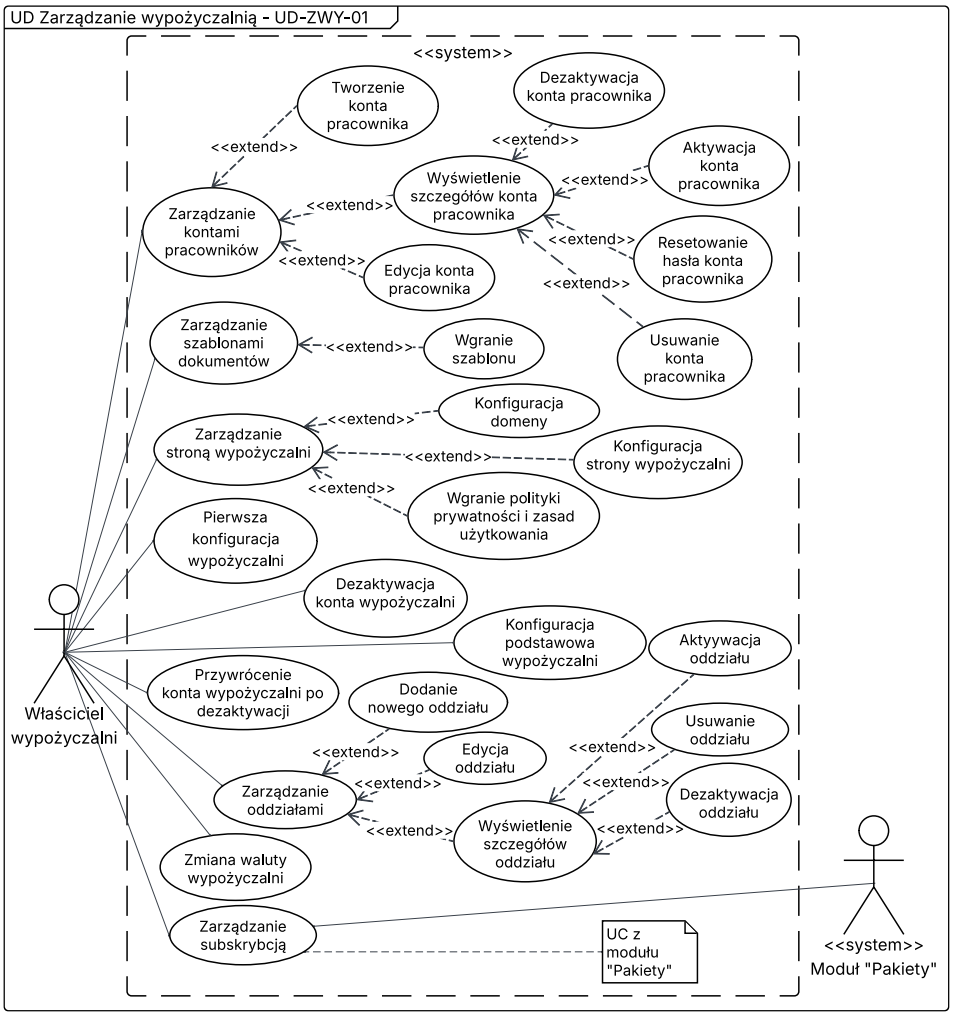
- **NF-PR-01**: Aplikacje przeglądarkowe systemu działają poprawnie w aktualnych wersjach przeglądarek Microsoft Edge, Google Chrome oraz Safari (stan na czerwiec 2026).
    - **NF-PR-02:** Aplikacja mobilna jest zbudowana w technologii wieloplatformowej i jest przeznaczona do działania na urządzeniach z systemem Android w wersji co najmniej 7.0 oraz iOS w wersji co najmniej 18.0.
    - **NF-PR-03:** Komponenty serwerowe systemu są uruchamiane w kontenerach, co eliminuje bezpośrednią zależność od konfiguracji sprzętowej danego serwera. Takie rozwiązanie ułatwia testowanie oraz produkcyjne wdrożenie systemu.
    - **NF-PR-04:** System udostępnia udokumentowany interfejs REST, który umożliwia współpracę z systemami zewnętrznymi.

5.5. Przypadki użycia

W tym podrozdziale przedstawiono diagramy przypadków użycia opracowane dla wszystkich modułów systemu Wellhicle. Każdy diagram odwzorowuje zbiór operacji dostępnych dla zdefiniowanych aktorów w ramach danego obszaru funkcjonalnego. Stanowiły one punkt wyjścia dla dalszych prac projektowych i implementacyjnych.

5.5.1. Zarządzanie wypożyczalnią

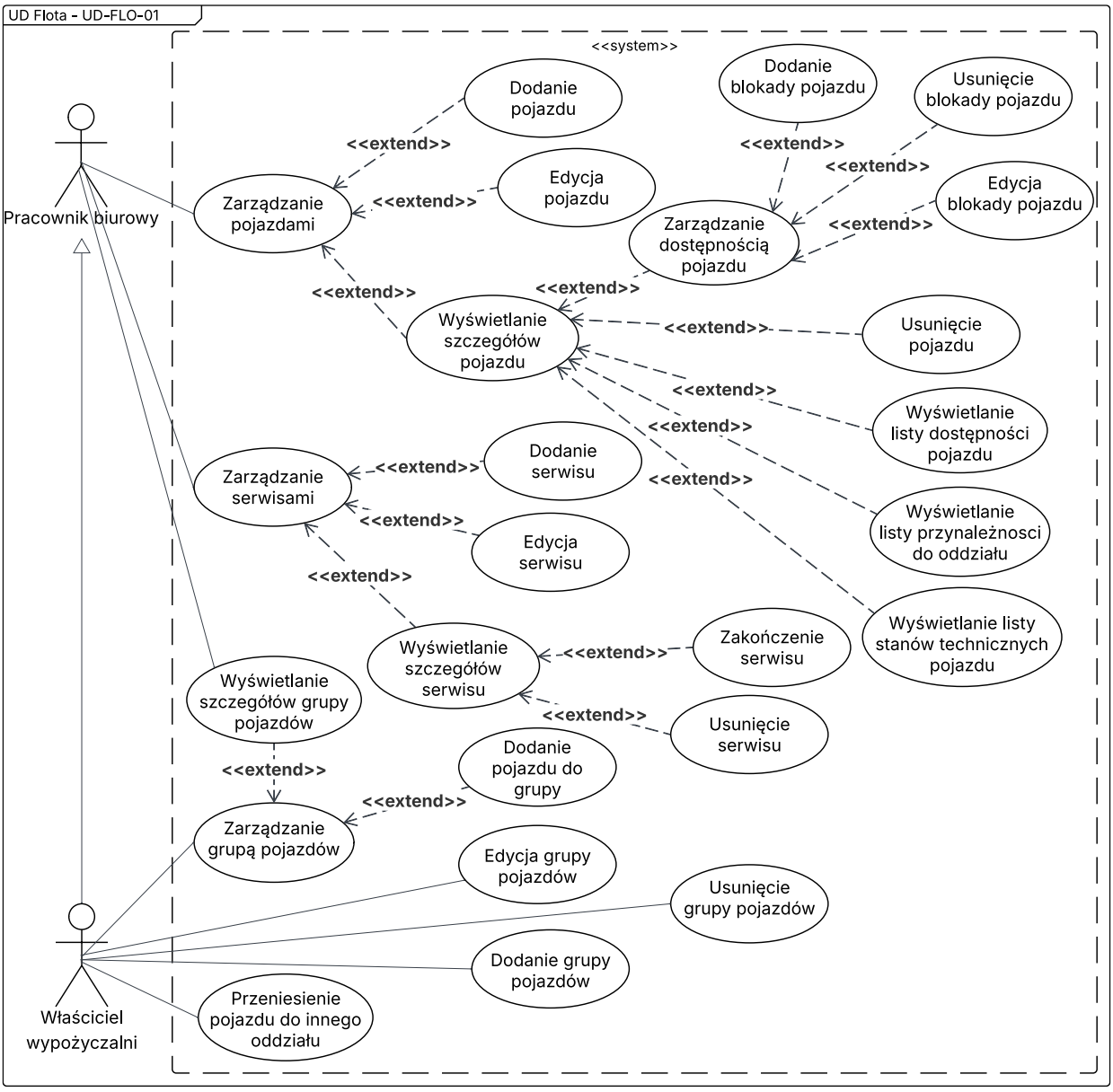
Diagram przypadków użycia modułu zarządzania wypożyczalnią przedstawia operacje dostępne dla właściciela wypożyczalni w zakresie konfiguracji i administrowania instancją wypożyczalni. Szczegółowy opis funkcjonalności zawarto w podpunkcie wymagań funkcjonalnych *(5.3.1)*. *Rysunek 17* przedstawia opisany diagram.



Rysunek 17 Diagram przypadków użycia modułu – zarządzanie wypożyczalnią. Źródło: opracowanie własne.

5.5.2. Flota

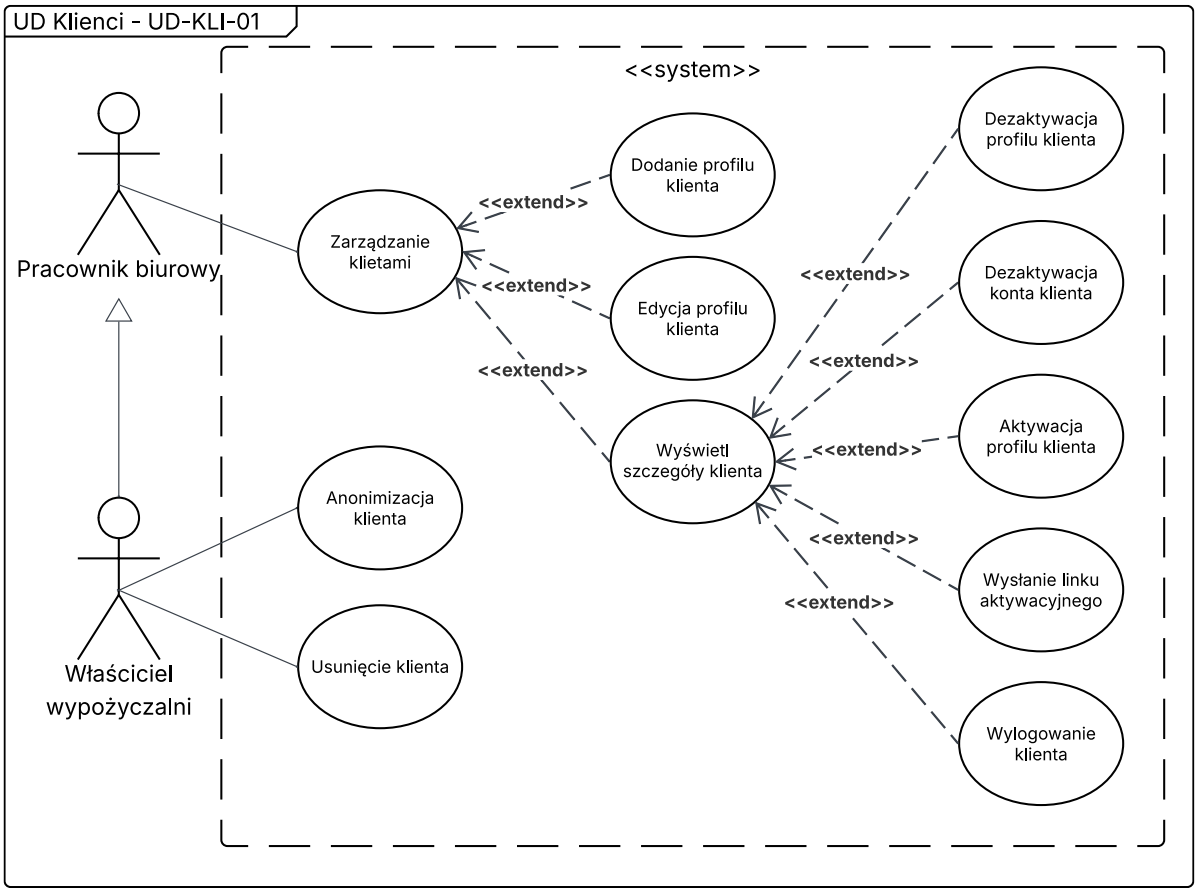
Rysunek 18 przedstawia diagram przypadków użycia modułu floty, obejmujący funkcjonalności zarządzania pojazdami dostępnymi w bazie wypożyczalni. Szczegółowy opis funkcjonalności modułu wraz z powiązanymi regułami biznesowymi zawarto w podpunkcie *5.3.2.2*.



Rysunek 18 Diagram przypadków użycia modułu – flota. Źródło: opracowanie własne.

5.5.3. Klienci

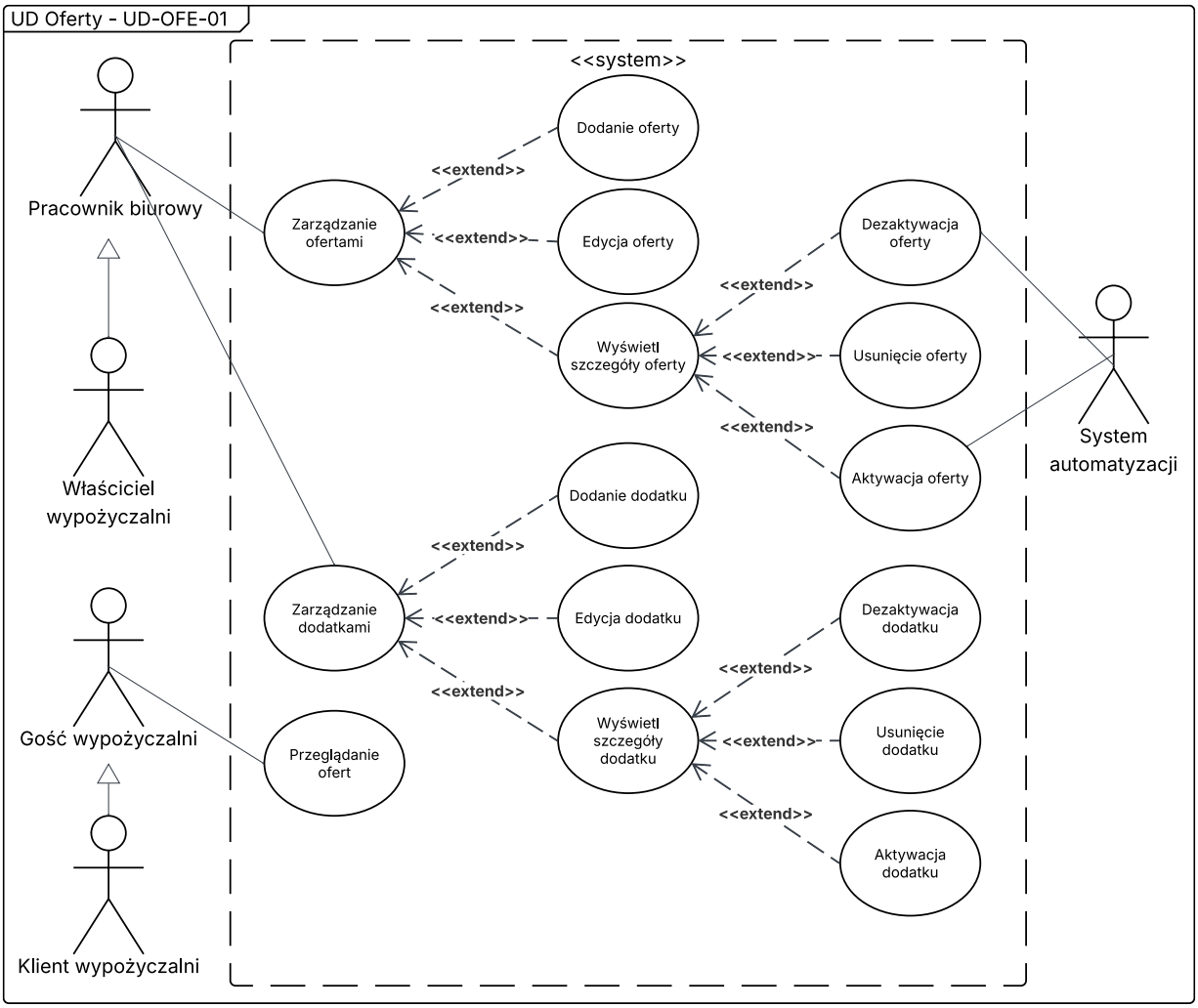
Rysunek 19 przedstawia diagram przypadków użycia modułu klientów, obejmujący funkcjonalności zarządzania bazą adresową wypożyczalni. Diagram uwzględnia m.in.: operacje rejestracji, edycji danych oraz zarządzania kontem klienta. Pełna specyfikacja wymagań funkcjonalnych modułu opisana została w podpunkcie *5.3.3.2*.



Rysunek 19 Diagram przypadków użycia modułu – klienci. Źródło: opracowanie własne.

5.5.4. Oferty

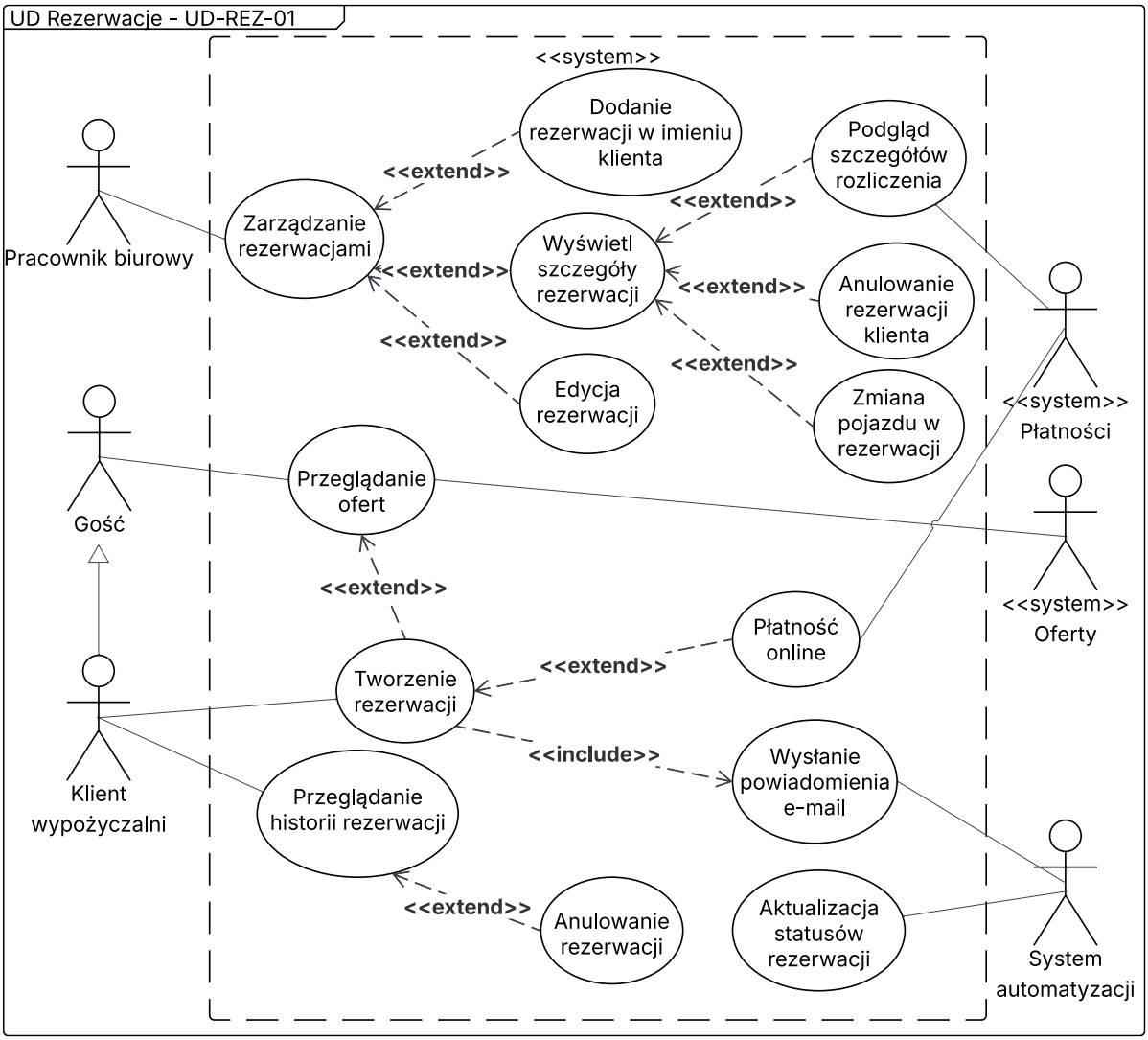
Rysunek 20 przedstawia diagram przypadków użycia modułu oferty, obejmujący operacje w zakresie zarządzania ofertami i dodatkami, wykonywane manualnie przez użytkownika oraz automatycznie przez system. Wymagania funkcjonalne modułu wraz z powiązanymi regułami biznesowymi szczegółowo opisano w podpunkcie *5.3.4.2*



Rysunek 20 Diagram przypadków użycia modułu – oferty. Źródło: opracowanie własne.

5.5.5. Rezerwacje

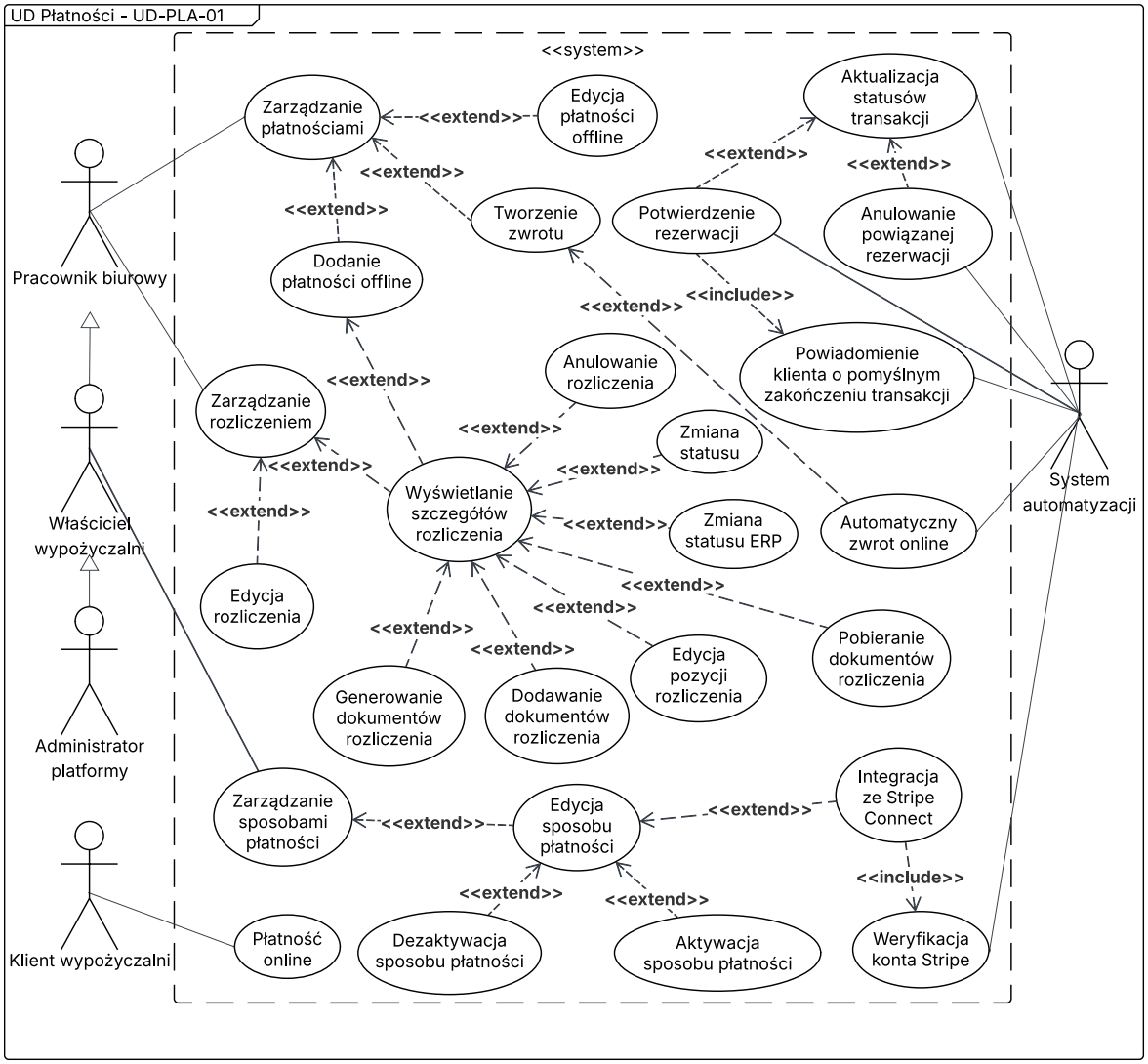
Moduł rezerwacji obejmuje funkcjonalności tworzenia i obsługi rezerwacji pojazdów, co przedstawiono na *Rysunek 21*. Diagram uwzględnia operacje dostępne zarówno dla pracownika biurowego, jak i klienta składającego rezerwację za pośrednictwem strony publicznej wypożyczalni. Szczegółowa specyfikacja wymagań funkcjonalnych modułu zawarta została w podpunkcie *5.3.5.2*



Rysunek 21 Diagram przypadków użycia modułu – rezerwacje. Źródło: opracowanie własne.

5.5.6. Płatności

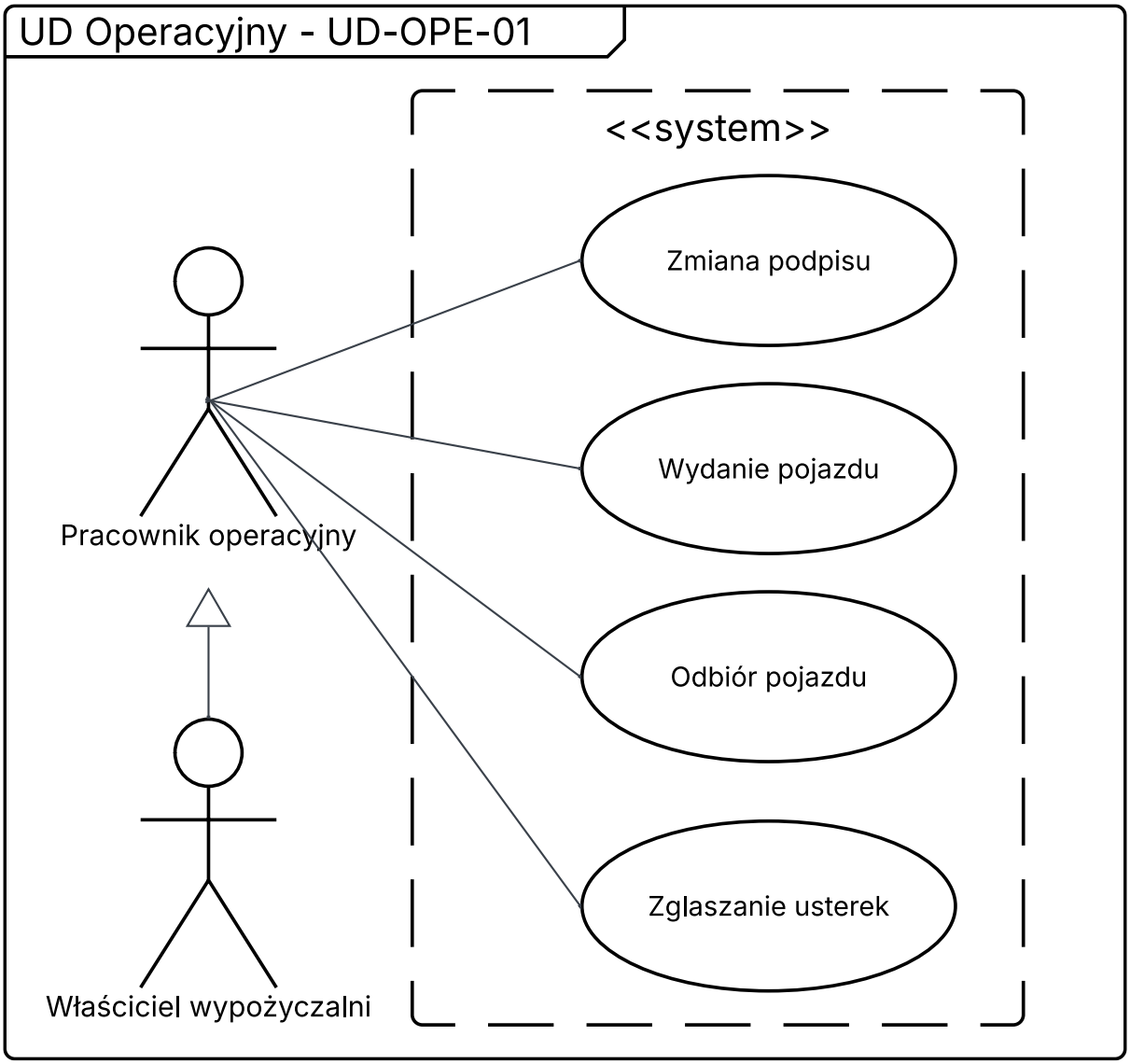
Na *Rysunek 22* przedstawiono diagram przypadków użycia modułu płatności, obejmujący funkcjonalności konfiguracji i obsługi metod płatności dostępnych w ramach wypożyczalni. Uwzględniono operacje zarządzania metodami płatności oraz rozliczania transakcji powiązanych z rezerwacjami. Wymagania funkcjonalne dotyczące płatności opisano w podpunkcie *5.3.6.2* oraz *5.3.7.2*.



Rysunek 22 Diagram przypadków użycia modułu – płatności. Źródło: opracowanie własne.

5.5.7. Operacyjny

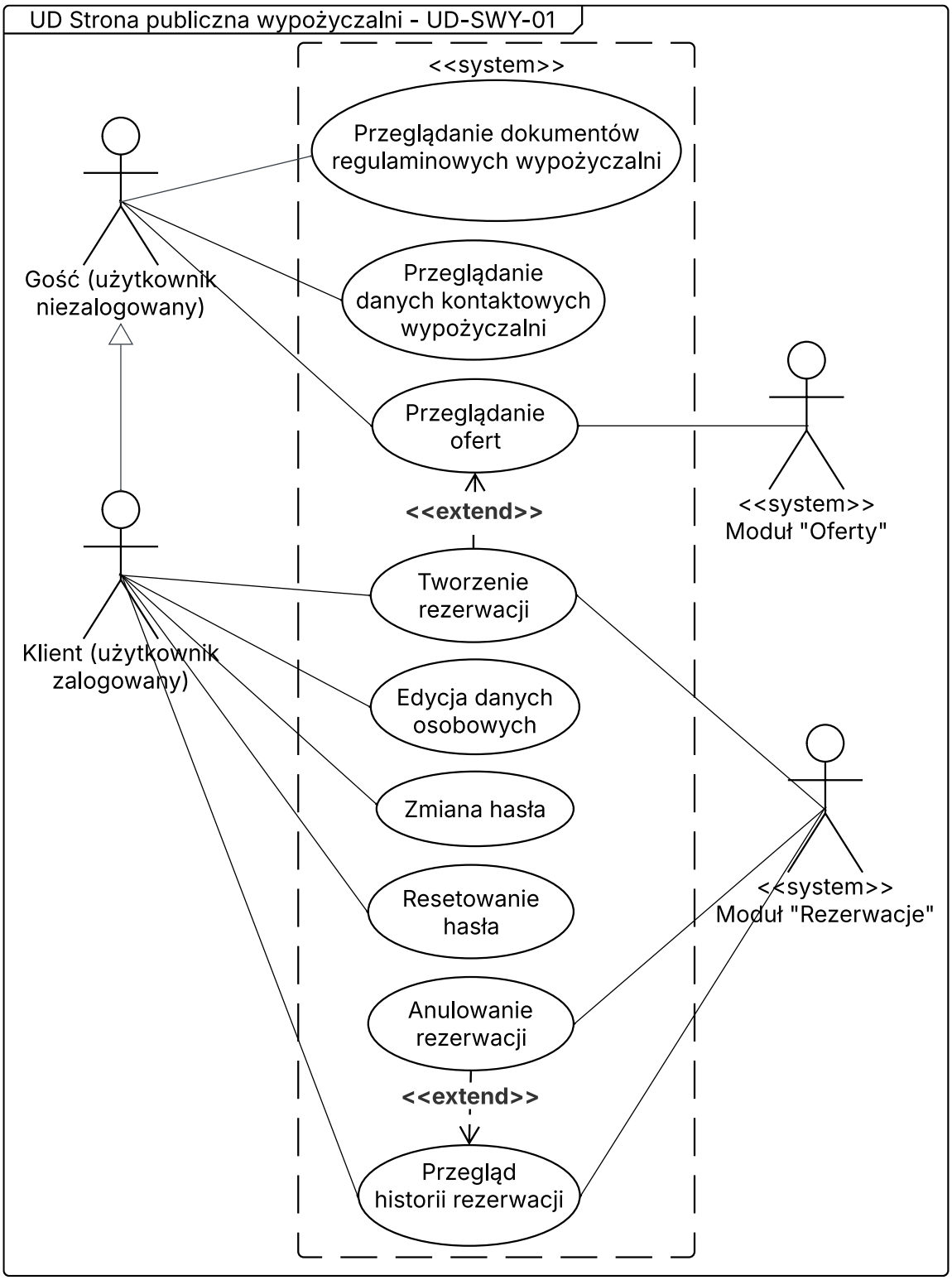
Aplikacja mobilna wspiera proces wydania oraz odbioru pojazdu udostępniając pracownikom operacyjnym zestaw funkcjonalności umożliwiających sprawną obsługę klienta w terenie. Przypadki użycia związane z tym modułem przedstawia *Rysunek 23*. Pełna specyfikacja wymagań funkcjonalnych modułu operacyjnego zawarta została w podpunkcie *5.3.8.2.*



Rysunek 23 Diagram przypadków użycia modułu – operacyjny. Źródło: opracowanie własne.

5.5.8. Strona publiczna wypożyczalni

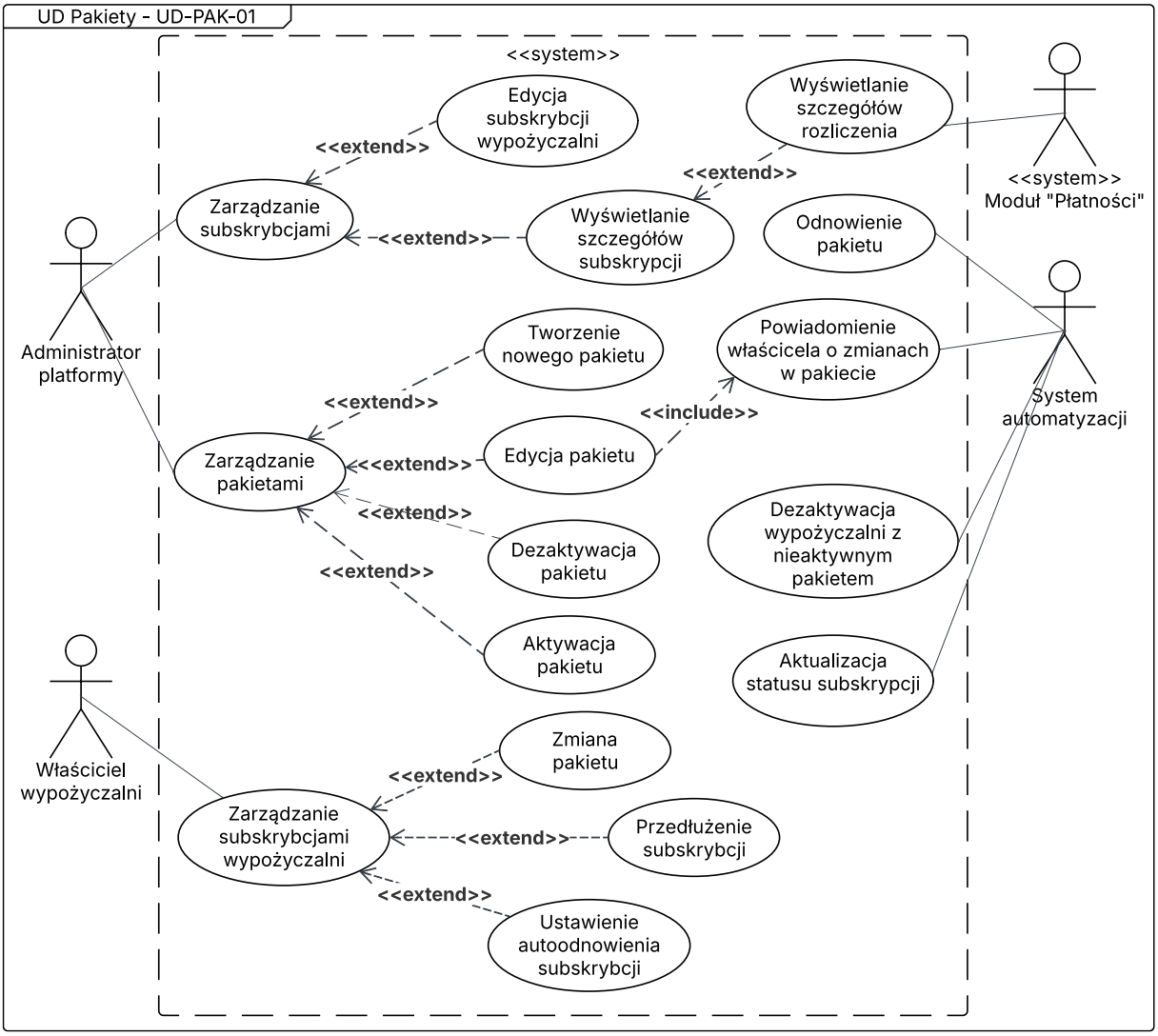
Rysunek 24 przedstawia diagram przypadków użycia modułu strony publicznej wypożyczalni, stanowiącej interfejs dostępny dla jej klientów. Szczegółowa specyfikacja wymagań funkcjonalnych modułu opisana została w podpunkcie *5.3.9.2.*



Rysunek 24 Diagram przypadków użycia modułu – strona publiczna wypożyczalni. Źródło: opracowanie własne.

5.5.9. Pakiety

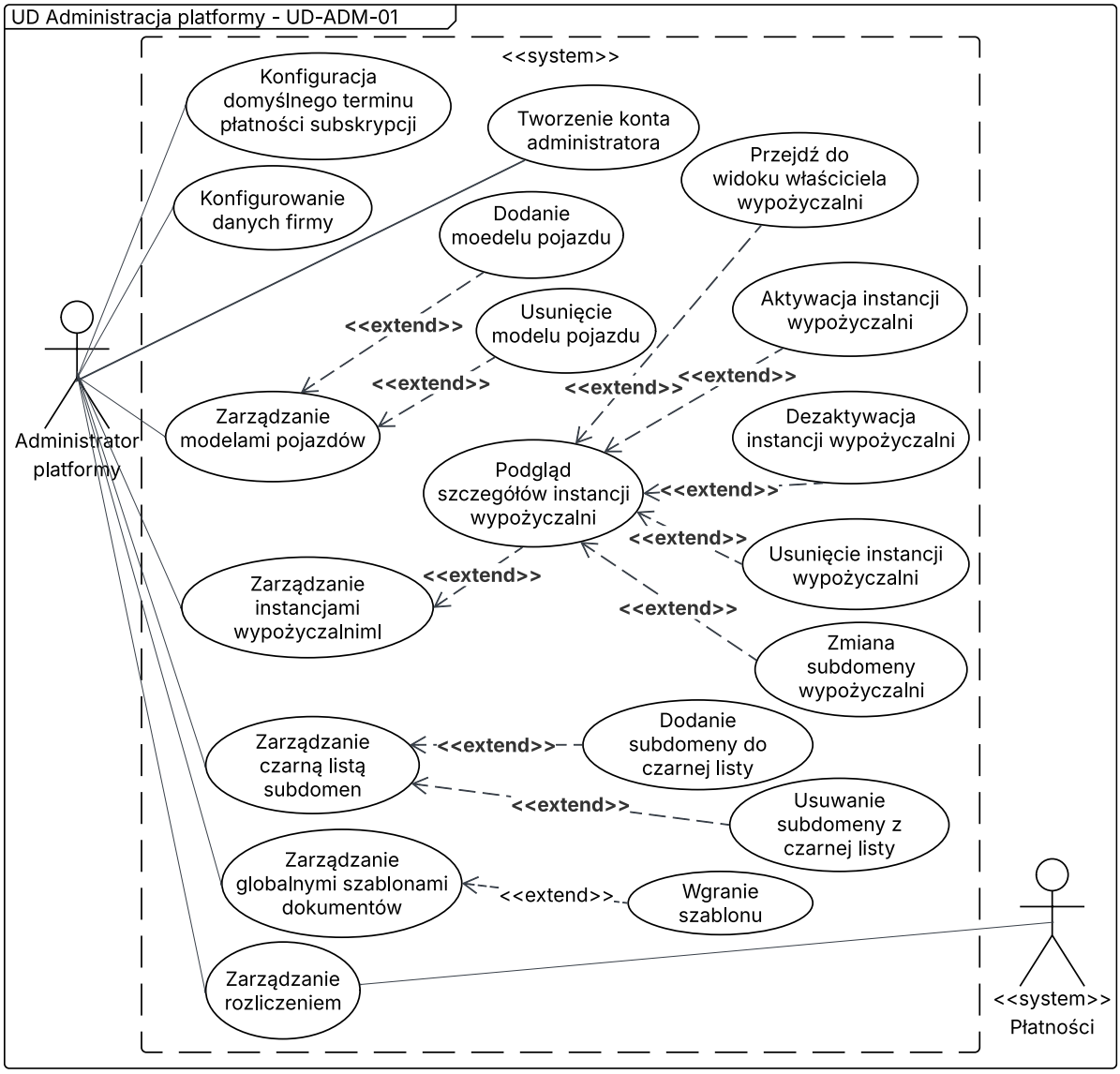
Rysunek 25 przedstawia diagram przypadków użycia modułu pakietów, obejmujący funkcjonalności zarządzania planami subskrypcyjnymi dostępnymi dla właścicieli wypożyczalni. Diagram obejmuje operacje przeglądania, wyboru oraz zmiany pakietu usługowego przez właścicieli oraz możliwości zarządzania nimi przez administratora platformy Wellhicle. Wymagania funkcjonalne modułu szczegółowo opisano w podpunkcie *5.3.10.2*.



Rysunek 25 Diagram przypadków użycia modułu – pakiety. Źródło: opracowanie własne.

5.5.10. Administracja platformy

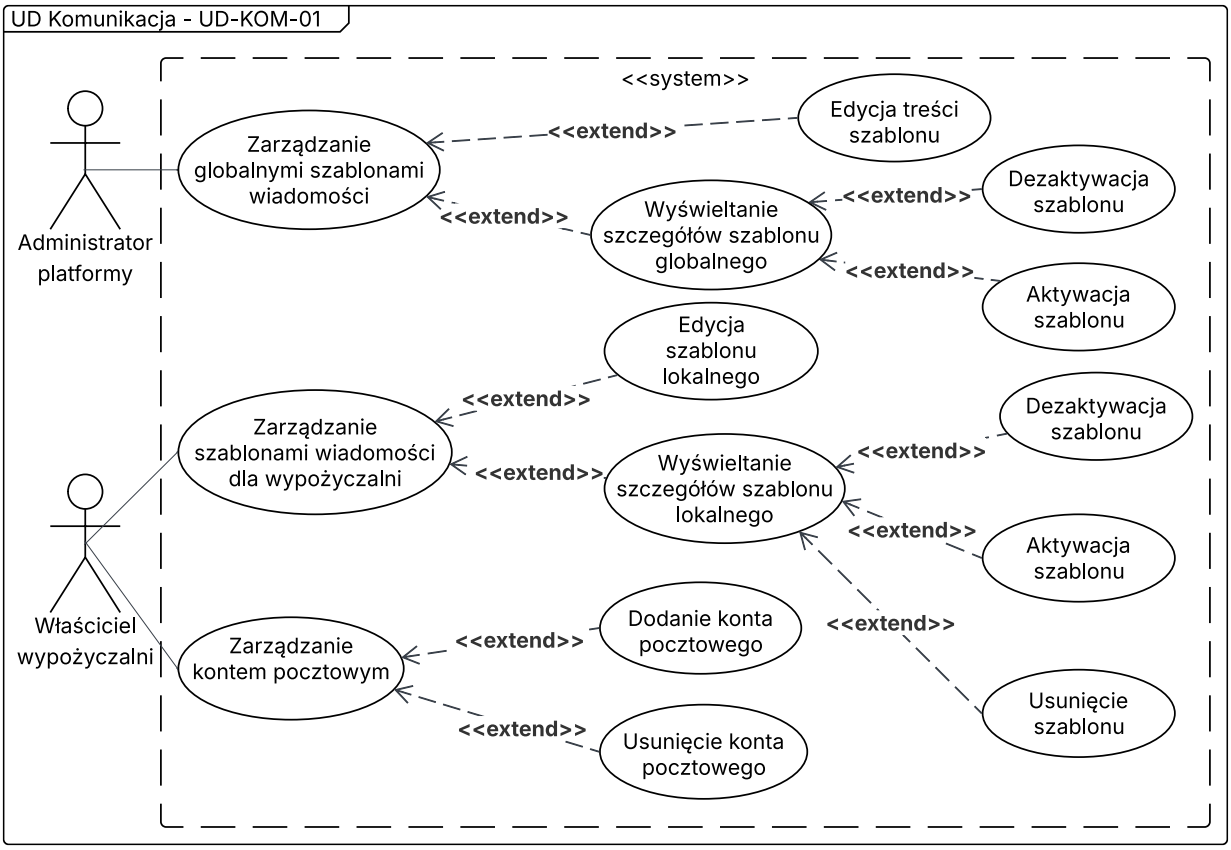
Moduł administracji platformy, przedstawiony na *Rysunek 26*, obejmuje funkcjonalności dostępne dla administratora zarządzającego całą platformą systemu Wellhicle. Zawarto w nim operacje nadzoru nad instancjami wypożyczalni, zarządzania subdomenami oraz bazą dostępnych modeli pojazdów. Pełna specyfikacja wymagań funkcjonalnych modułu zawarta została w podpunkcie *5.3.11.2.*



Rysunek 26 Diagram przypadków użycia modułu – administracja platformy. Źródło: opracowanie własne.

5.5.11. Komunikacja

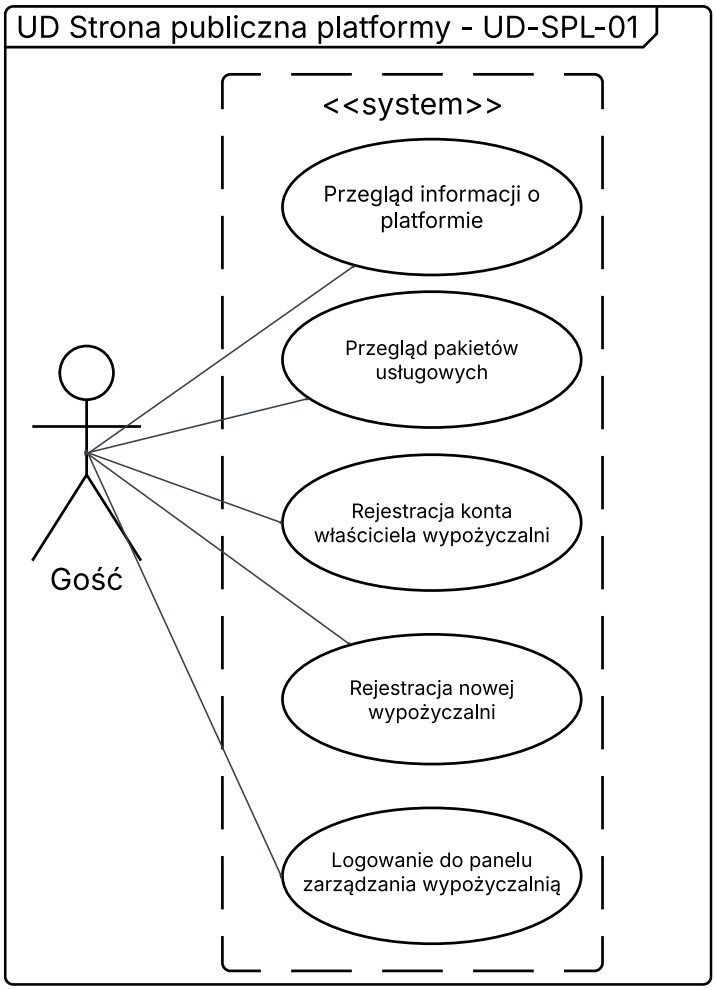
Rysunek 27 przedstawia diagram przypadków użycia modułu komunikacji, obejmujący funkcjonalności zarządzania szablonami wiadomości poczty elektronicznej. Wymagania funkcjonalne modułu szczegółowo opisano w podpunkcie *5.3.12.2.*



Rysunek 27 Diagram przypadków użycia modułu – komunikacja. Źródło: opracowanie własne.

5.5.12. Strona publiczna platformy

Na *Rysunek 28* przedstawiono diagram przypadków użycia modułu strony publicznej platformy, skierowanej do potencjalnych klientów zainteresowanych założeniem własnej wypożyczalni. Diagram obejmuje operacje przeglądania dostępnych pakietów usługowych oraz rejestracji nowej wypożyczalni. Szczegółowa specyfikacja wymagań funkcjonalnych modułu zawarta została w podpunkcie *5.3.13.2*.



Rysunek 28 Diagram przypadków użycia modułu – strona publiczna platformy. Źródło: opracowanie własne.

5.6. Wybrane scenariusze i diagramy aktywności

W podrozdziale zawarto kilka przykładowych diagramów aktywności oraz odpowiadające im scenariusze przypadków użycia, ilustrujące przebieg poszczególnych operacji realizowanych w systemie Wellhicle. Diagramy wykorzystują generyczne wzorce aktywności opisane w punkcie *5.6.1* co pozwoliło na ograniczenie powtarzania wspólnych elementów dla wielu przepływów.

5.6.1. Generyczne wzorce aktywności

W niniejszym podpunkcie przedstawiono najbardziej powtarzalne wzorce aktywności występujące w analizie systemu Wellhicle. Nie obejmują one wszystkich diagramów przygotowanych dla całego systemu, lecz te fragmenty procesów, które pojawiają się wielokrotnie i mogą zostać opisane w sposób wspólny. Wyróżniono dwa główne rodzaje takich diagramów opisane w podpunktach: *5.6.1.1* oraz *5.6.1.2*.

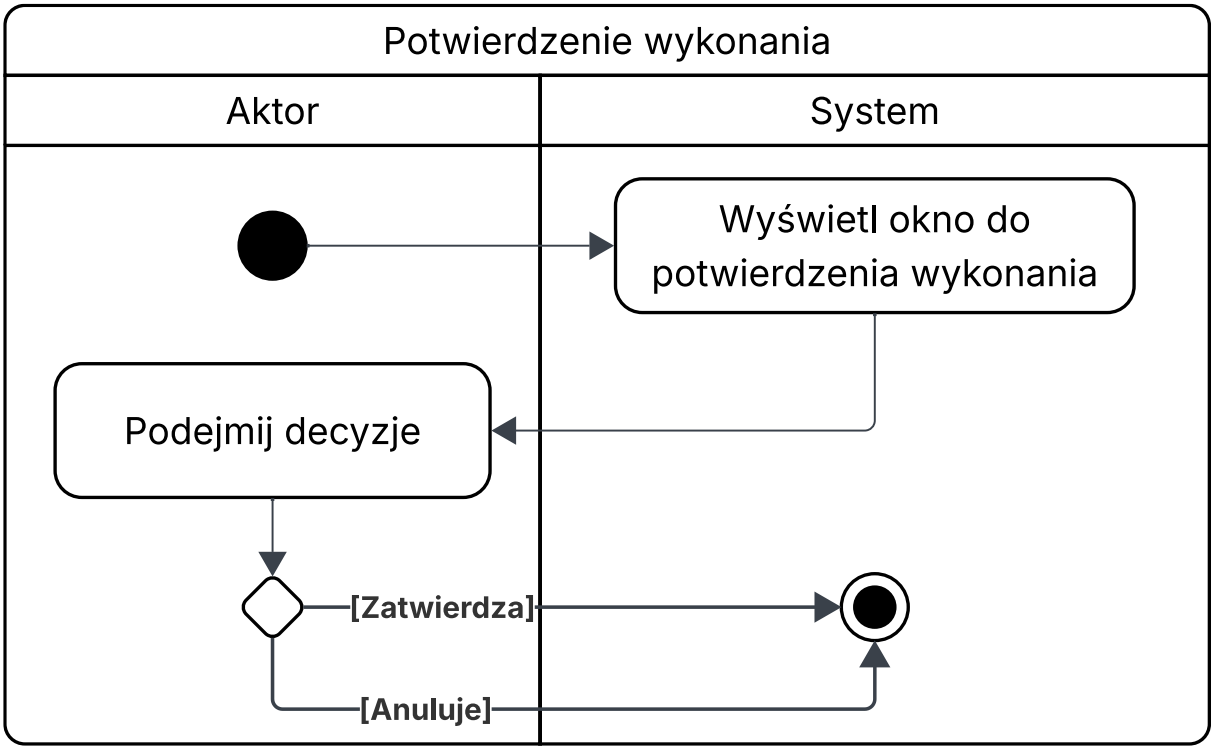
5.6.1.1. Wspólne diagramy pomocnicze

Diagramy pomocnicze przedstawiają przepływy wykorzystane w systemie w identycznej postaci w różnych modułach. W diagramach aktywności odwołania do tych przepływów oznaczono jako akcje wywołującą zdefiniowane zachowanie (ang. Call Behavior Action). Zgodnie z notacją UML,

ta akcja służy do wywołania wcześniej zdefiniowanego zachowania, które dla diagramu nadrzędnego stanowi pojedynczą czynność, choć wewnętrznie może obejmować bardziej złożony przebieg. Pozwala to na powielanie tego samego fragmentu procesu w różnych miejscach systemu bez konieczności ponownego definiowania jego zawartości [94].

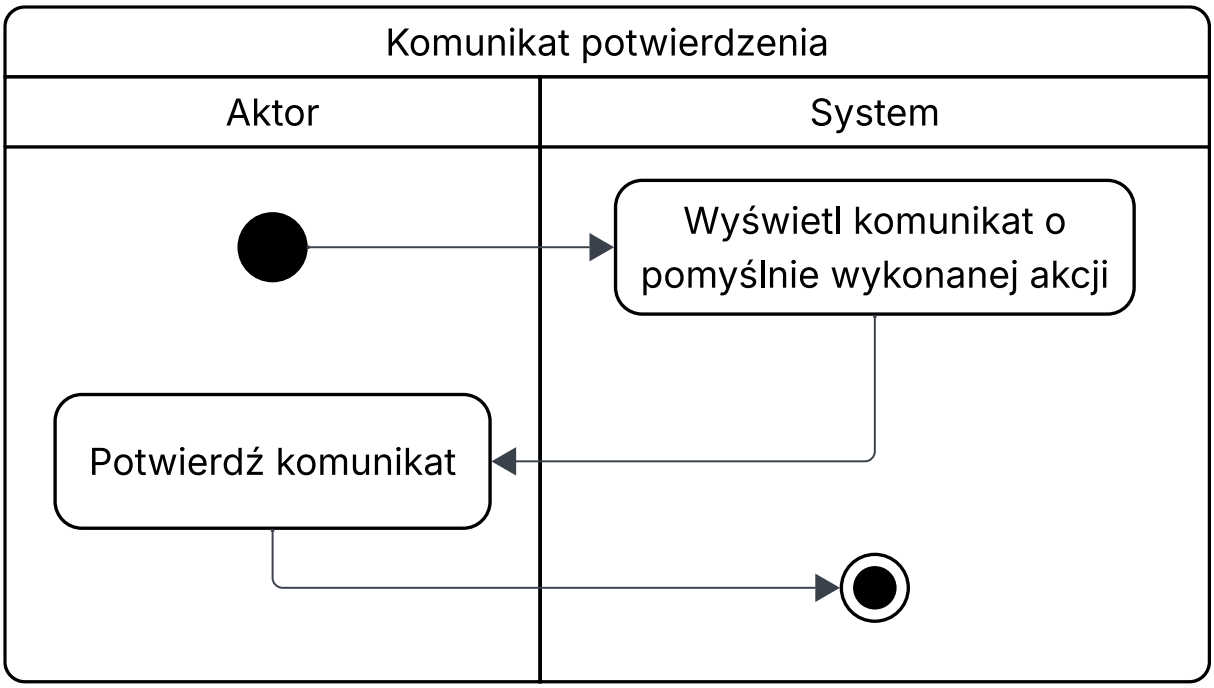
W tym przypadku nie są to uogólnienia różnych operacji, lecz konkretnie zaimplementowane w systemie przepływy pracy o jednolitej strukturze. Dotyczą one głównie sytuacji, gdy system musi uzyskać potwierdzenie od uczestnika procesu lub poinformować go o wyniku wykonanej czynności.

Przedstawiony na *Rysunek 29* diagram potwierdzenia wykonania opisuje sytuację, w której system wymaga dodatkowego zatwierdzenia operacji przez aktora przed jej finalizacją. Aktor otrzymuje możliwość zaakceptowania lub anulowania czynności. Rozwiązanie to stosuje się dla działań wpływających na stan danych bądź mających trwałe konsekwencje, takich jak usunięcie rekordu, aktywacja lub dezaktywacja obiektów. Diagram został użyty jako gotowy element przepływu pracy, a nie jedynie abstrakcyjny przykład.



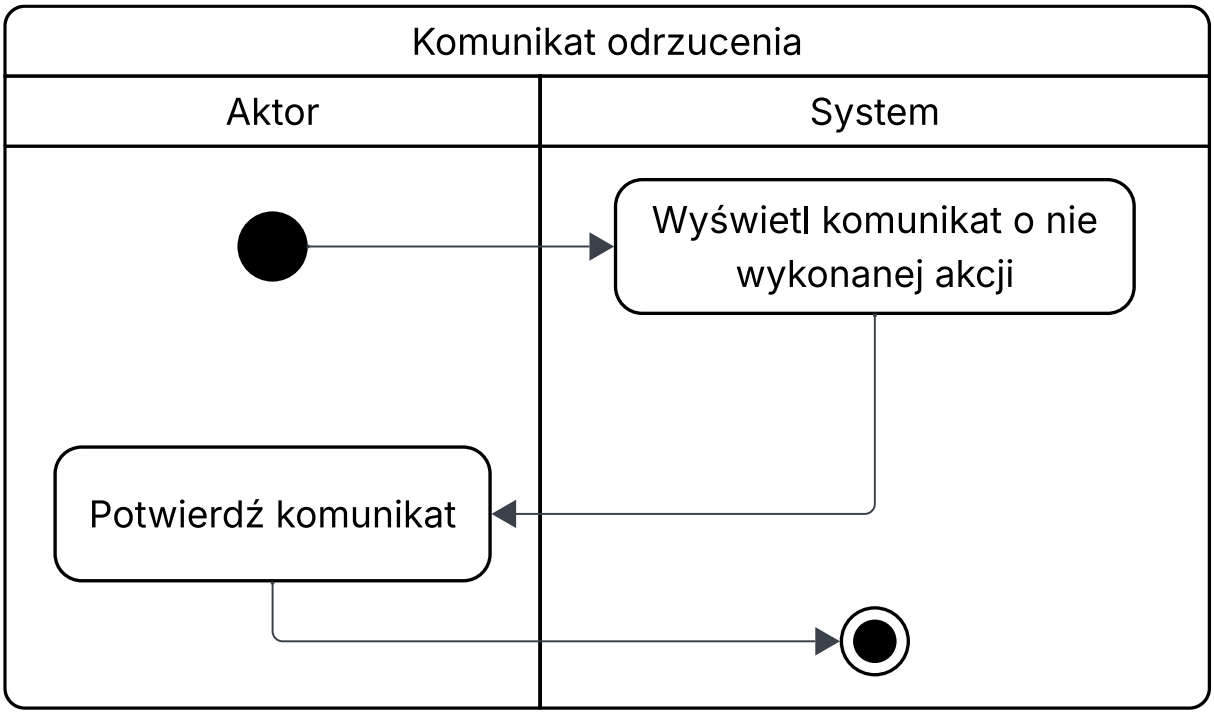
Rysunek 29 Diagram potwierdzenia wykonania. Źródło: opracowanie własne.

Diagram komunikatu potwierdzenia zaprezentowany na *Rysunek 30,* opisuje prostą reakcję systemu po pomyślnym wykonaniu czynności. System prezentuje użytkownikowi informację o powodzeniu operacji, a następnie kończy dany etap procesu. Diagram nie wprowadza dodatkowych decyzji ani zmian po stronie aktora – jego rolą jest jednoznaczne zamknięcie operacji dla użytkownika. Został zastosowany wszędzie tam, gdzie system musi potwierdzić poprawne zapisanie, modyfikację lub wykonanie danych.



Rysunek 30 Diagram komunikatu potwierdzenia. Źródło: opracowanie własne.

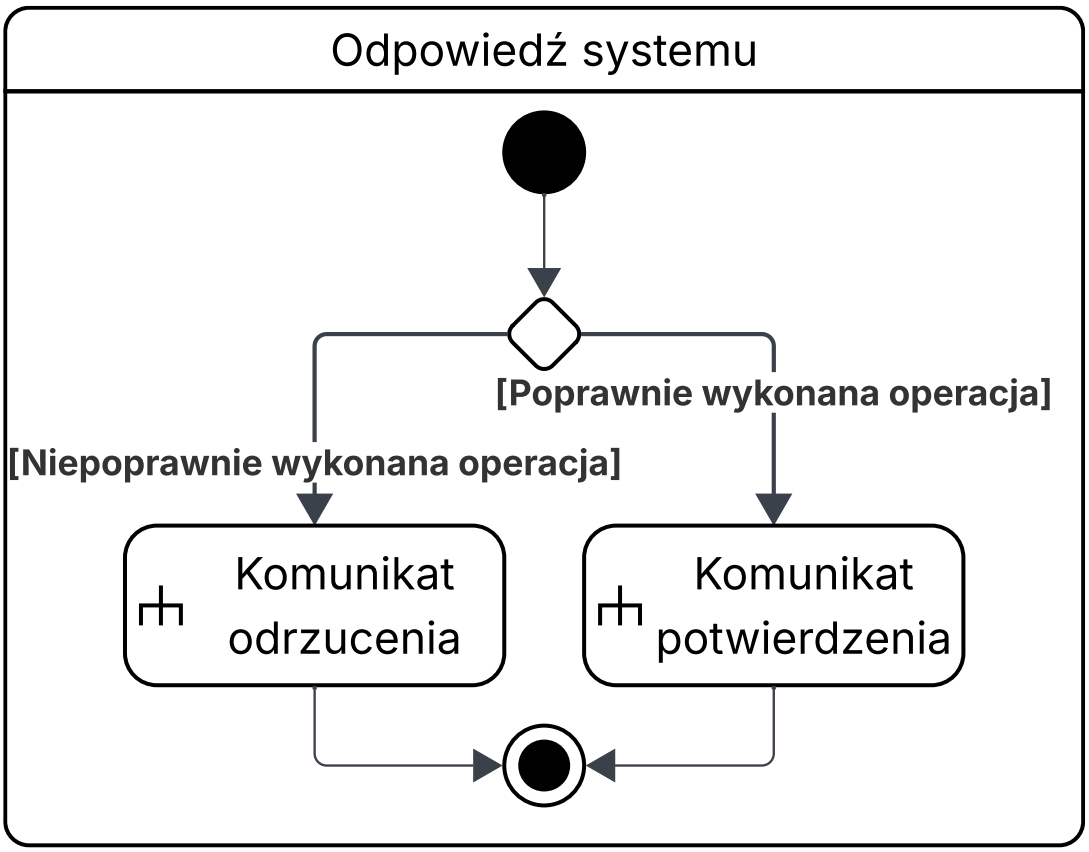
Analogicznie zamodelowany został diagram komunikatu odrzucenia przedstawiony na *Rysunek 31*, który opisuje reakcję systemu w sytuacji, gdy wykonanie czynności nie jest możliwe. Jego przebieg jest zbliżony do schematu odpowiadającego potwierdzeniu, jednakże różni się tym, że zamiast informować o pomyślnym wykonaniu operacji, system prezentuje użytkownikowi komunikat wskazujący na niewykonanie akcji.



Rysunek 31 Diagram komunikatu odrzucenia. Źródło: opracowanie własne.

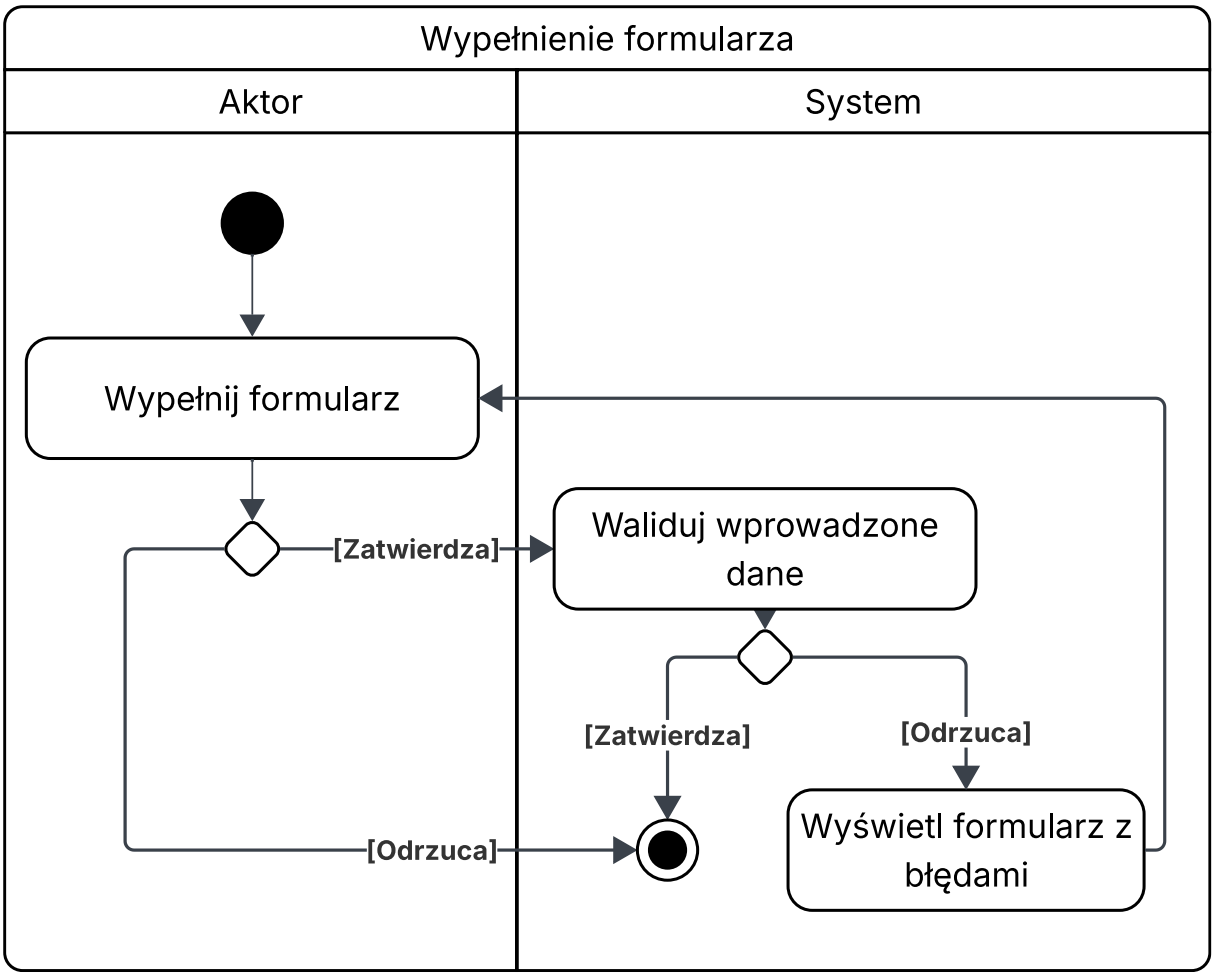
Zaprezentowany na *Rysunek 32* diagram odpowiedzi systemu przedstawia sposób finalizacji operacji po stronie aplikacji. System weryfikuje wynik wykonanej czynności i kieruje przepływ do jednego z dwóch komunikatów: potwierdzenia lub odrzucenia. W przypadku pomyślnego zakończenia operacji używany jest komunikat potwierdzający, który informuje użytkownika o sukcesie działania.

Jeśli warunki wykonania nie zostały spełnione, system generuje komunikat odrzucenia, wskazujący na brak realizacji akcji. Ten sam schemat zastosowano wielokrotnie, ponieważ sposób przekazywania informacji o wyniku operacji musi być jednolity we wszystkich częściach systemu.



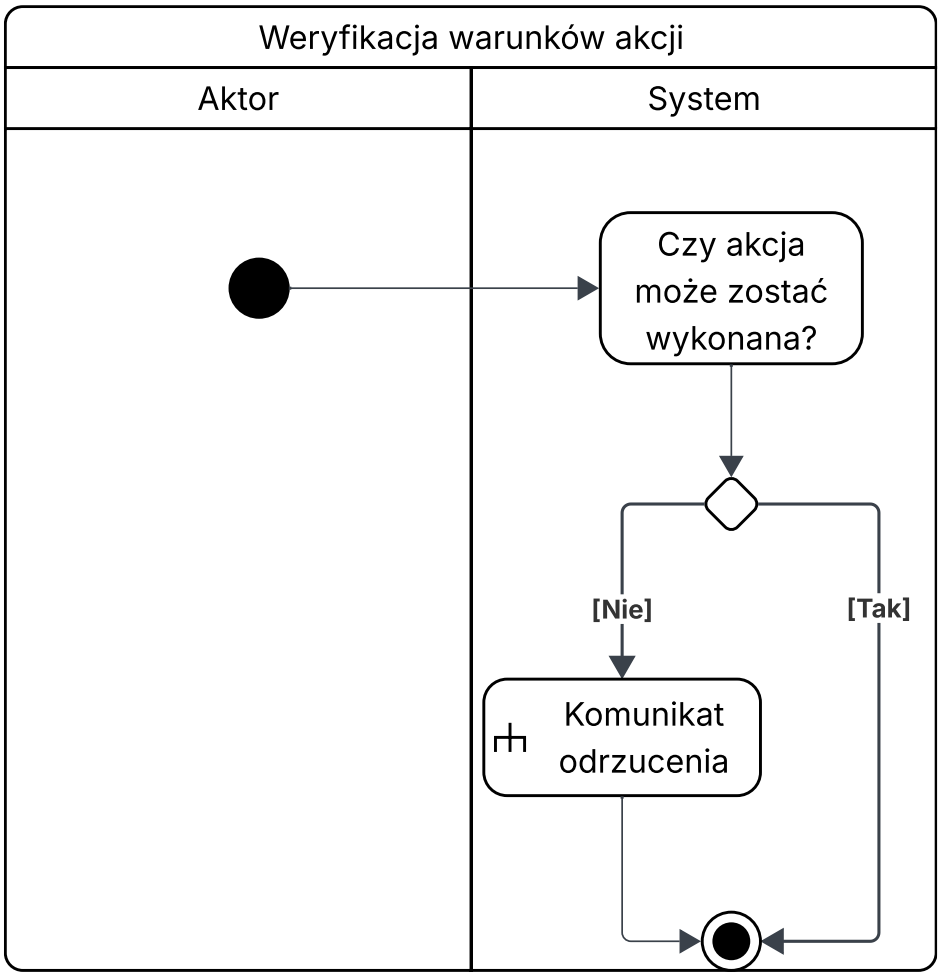
Rysunek 32 Diagram odpowiedzi systemu. Źródło: opracowanie własne.

Diagram wypełnienia formularza przedstawia proces obsługi formularza przez aktora i system. Aktor uzupełnia dane, a następnie może zatwierdzić lub odrzucić wprowadzone informacje. Po akceptacji system przeprowadza walidację danych – w zależności od jej wyniku kończy etap albo wyświetla formularz zawierający błędy do poprawy. Diagram ten został zastosowany we wszystkich obszarach, gdzie użytkownik wprowadza dane podlegające sprawdzeniu przed kontynuacją procesu. Diagram został zaprezentowany na *Rysunek 33*.



Rysunek 33 Diagram wypełnienia formularza. Źródło: opracowanie własne.

Diagram weryfikacji warunków akcji zaprezentowany na *Rysunek 34*, przedstawia sposób sprawdzenia, czy dana czynność może zostać wykonana. System ocenia warunki konieczne do przeprowadzenia operacji i kieruje przepływem procesu w zależności od wyniku tej oceny. Jeśli akcja jest dopuszczalna, następuje przejście do etapu wykonania właściwej operacji. W przypadku niespełnienia warunków system generuje komunikat odrzucenia i kończy daną ścieżkę postępowania. Diagram ten jest stosowany przy czynnościach zależnych od stanu obiektu lub jego powiązań z innymi danymi, co pozwala na elastyczne dostosowanie przepływu procesów do specyficznych wymagań biznesowych.



Rysunek 34 Diagram weryfikacji warunków akcji. Źródło: opracowanie własne.

5.6.1.2. Wzorce operacji na obiektach

Wzorce obsługi obiektów systemowych przedstawiają wspólny sposób realizacji podstawowych operacji wykonywanych na danych systemu Wellhicle. W przeciwieństwie do pojedynczych diagramów pomocniczych, nie są one stosowane w identycznej postaci dla każdego obiektu, lecz stanowią uogólnienie schematów przygotowanych dla konkretnych typów danych.

Wspólny dla tych schematów jest ogólny przebieg czynności wykonywanej na obiekcie. Aktor inicjuje operację, a system prowadzi proces zgodnie z przyjętymi regułami analitycznymi, takimi jak konieczność uzupełnienia danych, potwierdzenia decyzji lub sprawdzenia warunków wykonania akcji. Różnice między diagramami dla poszczególnych obiektów dotyczą przede wszystkim rodzaju przetwarzanych danych, specyficznych warunków biznesowych oraz skutków realizacji operacji.

W celu usprawnienia prezentacji analizy operacji w pracy, przyjęto podejście polegające na przedstawieniu ich za pomocą wspólnych wzorców. Zamieszczenie wszystkich diagramów dla każdego obiektu prowadziłaby do powtarzania podobnych przepływów, różniących się jedynie drobnymi szczegółami specyficznymi dla danego elementu.

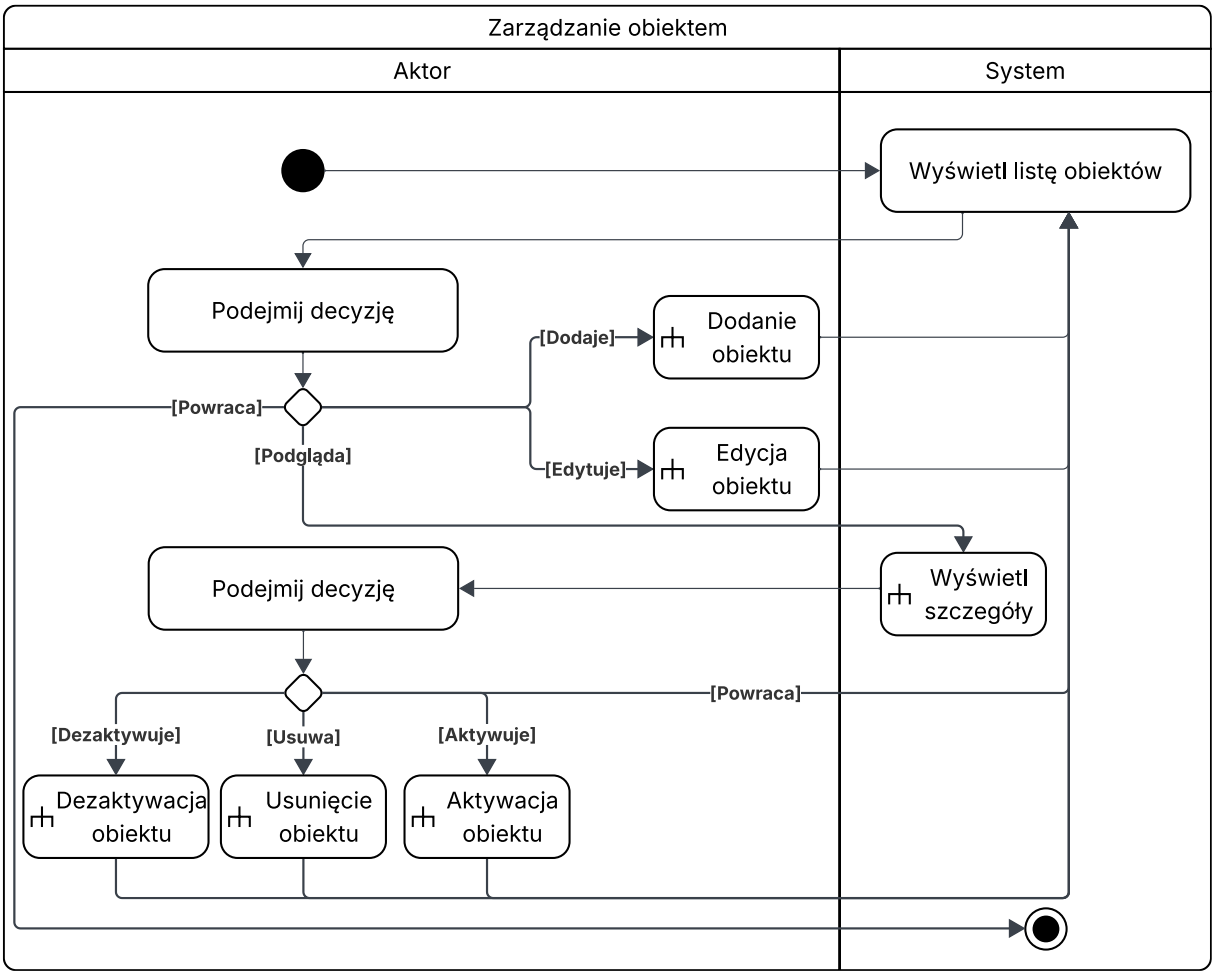
Dzięki takiemu rozwiązaniu możliwe jest skupienie uwagi na kluczowych aspektach omawianych procesów, bez konieczności powtarzania diagramów zawierających jedynie nieznaczne modyfikacje.

Diagram zarządzania obiektem zaprezentowany na *Rysunek 35,* przedstawia ogólny sposób pracy z listą obiektów w systemie Wellhicle. Aktor rozpoczyna proces, wyświetlając listę dostępnych

elementów, a następnie dokonuje wyboru jednej z możliwych akcji. System oferuje operacje dopasowane do typu obiektu oraz jego aktualnego stanu, takie jak:

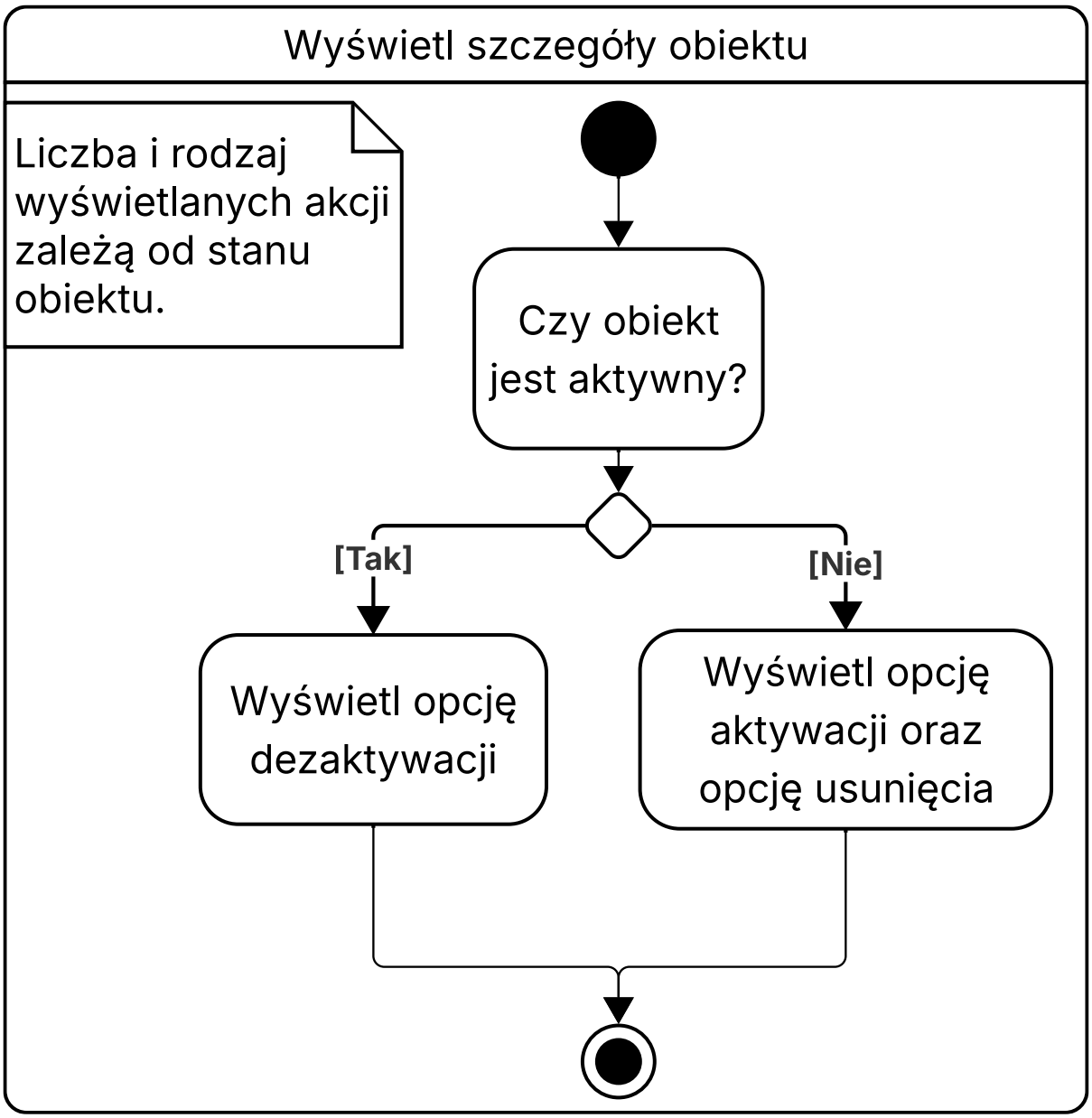
- dodanie nowego rekordu
- edycja istniejącego
- wyświetlenie szczegółów
- zmiana statusu (aktywacja/dezaktywacja)
- usunięcie.

Diagram pełni funkcję nadrzędną w kontekście innych operacji na obiektach, stanowiąc podstawę dla dalszych działań użytkownika systemu.



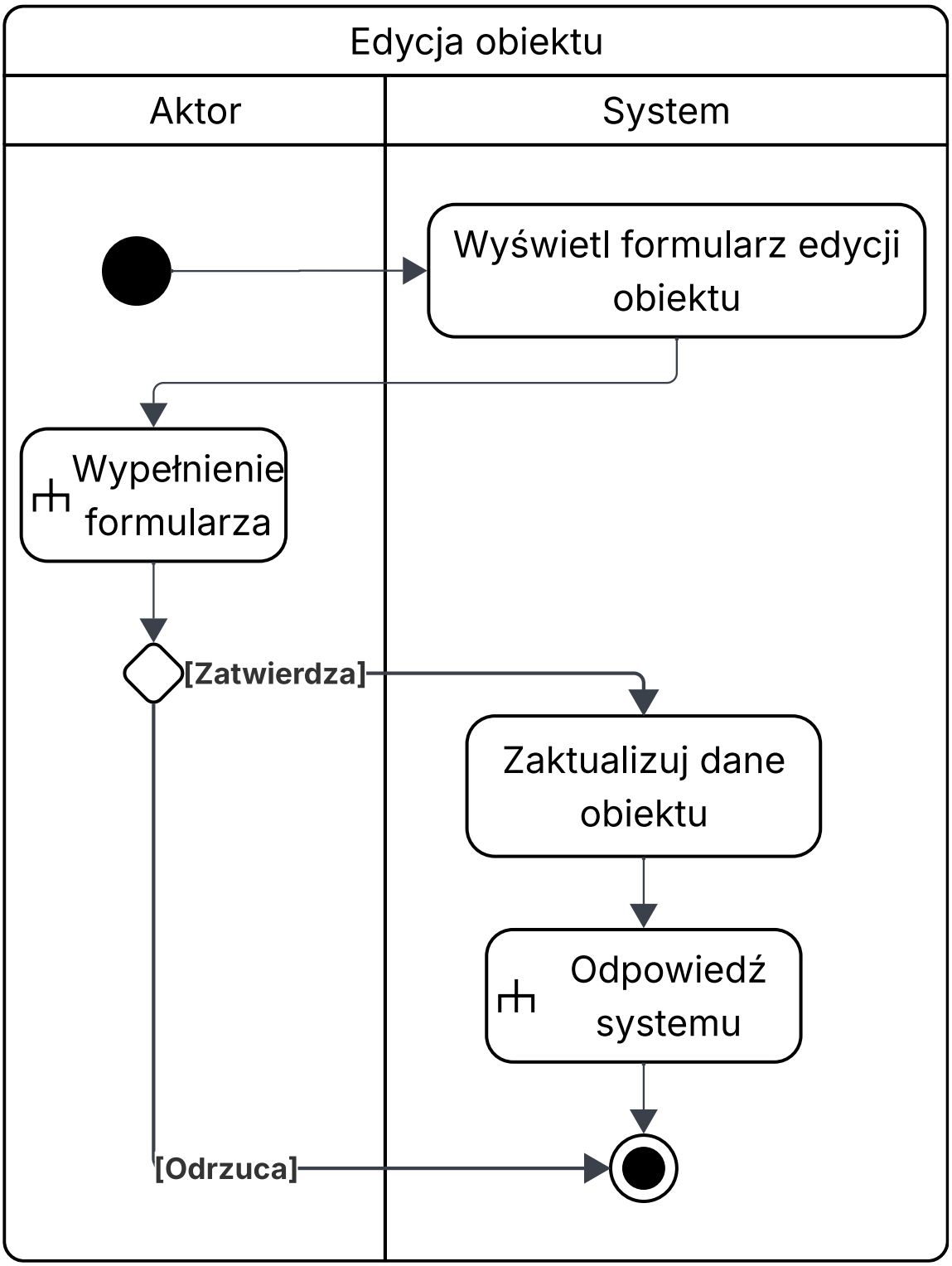
Rysunek 35 Diagram zarządzania obiektem. Źródło: opracowanie własne.

Diagram wyświetlenia szczegółów obiektu prezentuje podgląd danych wybranego rekordu, nie modyfikując jego stanu. Umożliwia aktorowi zapoznanie się z właściwościami obiektu oraz podjęcie decyzji dotyczącej jego przyszłych działań. System dostosowuje prezentowane informacje do typu danych, a następnie udostępnia akcje zależne od aktualnego stanu obiektu: dla obiektów aktywnych oferuje możliwość dezaktywacji, natomiast w przypadku obiektów nieaktywnych – opcję reaktywacji lub usunięcia. Zarówno zakres prezentowanych danych, jak i dostępne akcje są zależne od modułu, co zostało uogólnione w przedstawionym przebiegu czynności. Diagram został przedstawiony na *Rysunek 36.*



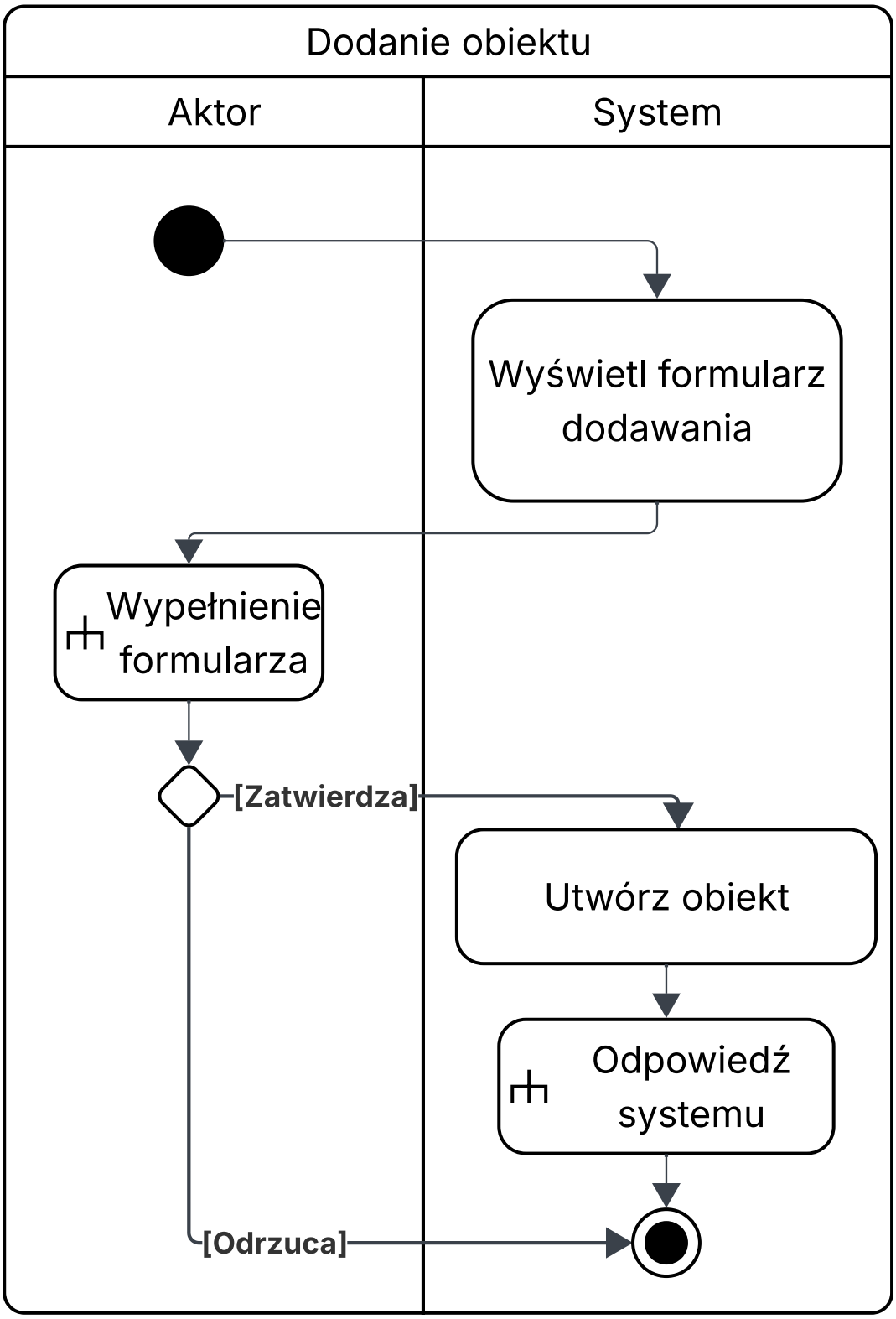
Rysunek 36 Diagram wyświetlenia szczegółów obiektu. Źródło: opracowanie własne.

Diagram edycji obiektu przedstawia proces modyfikacji danych istniejącego elementu systemu. Proces rozpoczyna się od wyświetlenia formularza edycji, który umożliwia aktorowi zmianę dotychczasowych wartości. Po zatwierdzeniu modyfikacji system dokonuje aktualizacji danych obiektu i przekazuje odpowiedź zwrotną, co stanowi standardowy schemat postępowania niezależnie od typu edytowanego elementu. Diagram przedstawiony na *Rysunek 37*, ilustruje więc uniwersalny mechanizm aktualizacji danych obiektów w systemie Wellhicle.



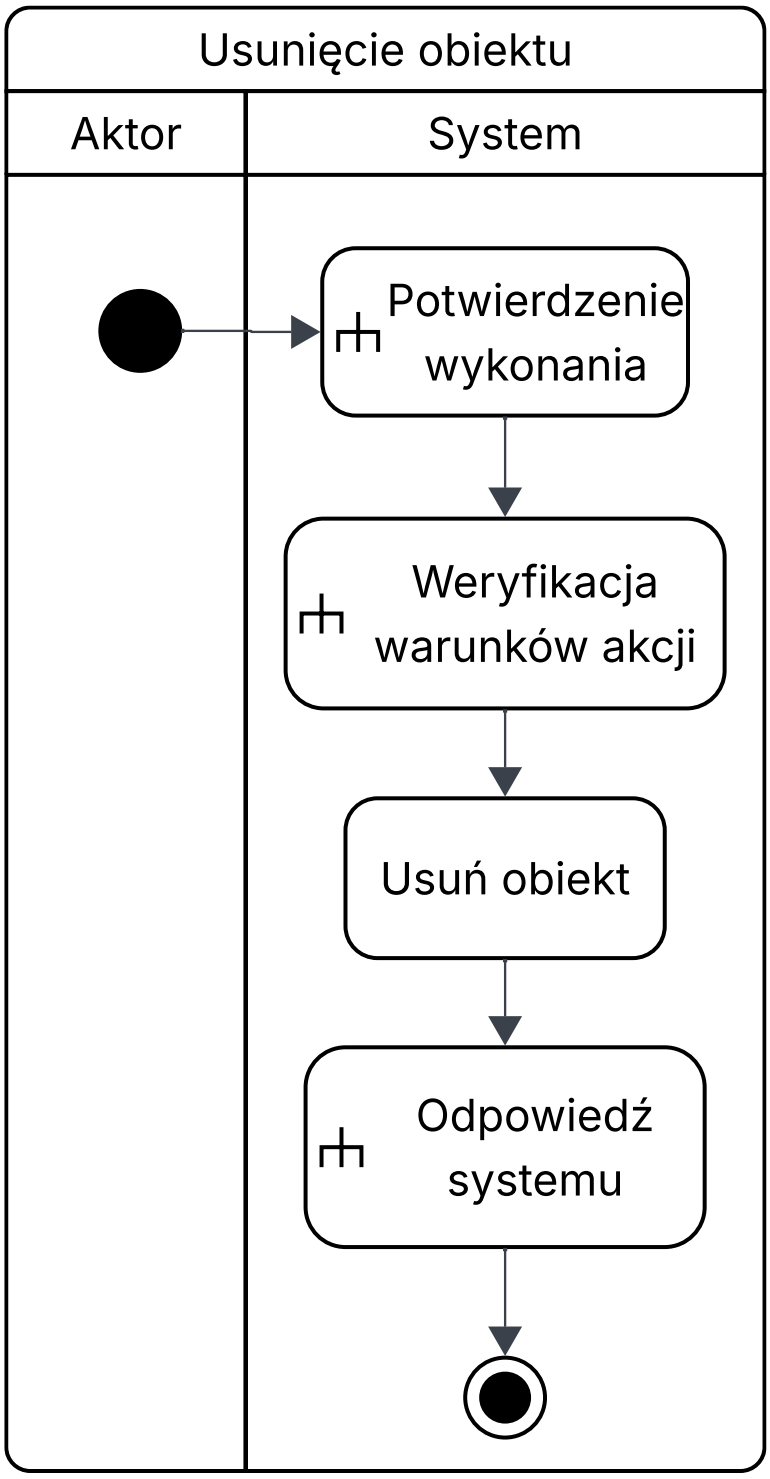
Rysunek 37 Diagram edycji obiektu. Źródło: opracowanie własne.

Diagram dodania obiektu przedstawiony na *Rysunek 38*, jest niemal identyczny w stosunku do diagramu edycji, ponieważ zarówno w jednym, jak i drugim przypadku aktor korzysta z formularza, a system przetwarza wprowadzone dane i odpowiada informacją o wyniku operacji. Kluczowa różnica polega na celu działania, ponieważ w diagramie dodawania obiektu tworzony jest nowy rekord.



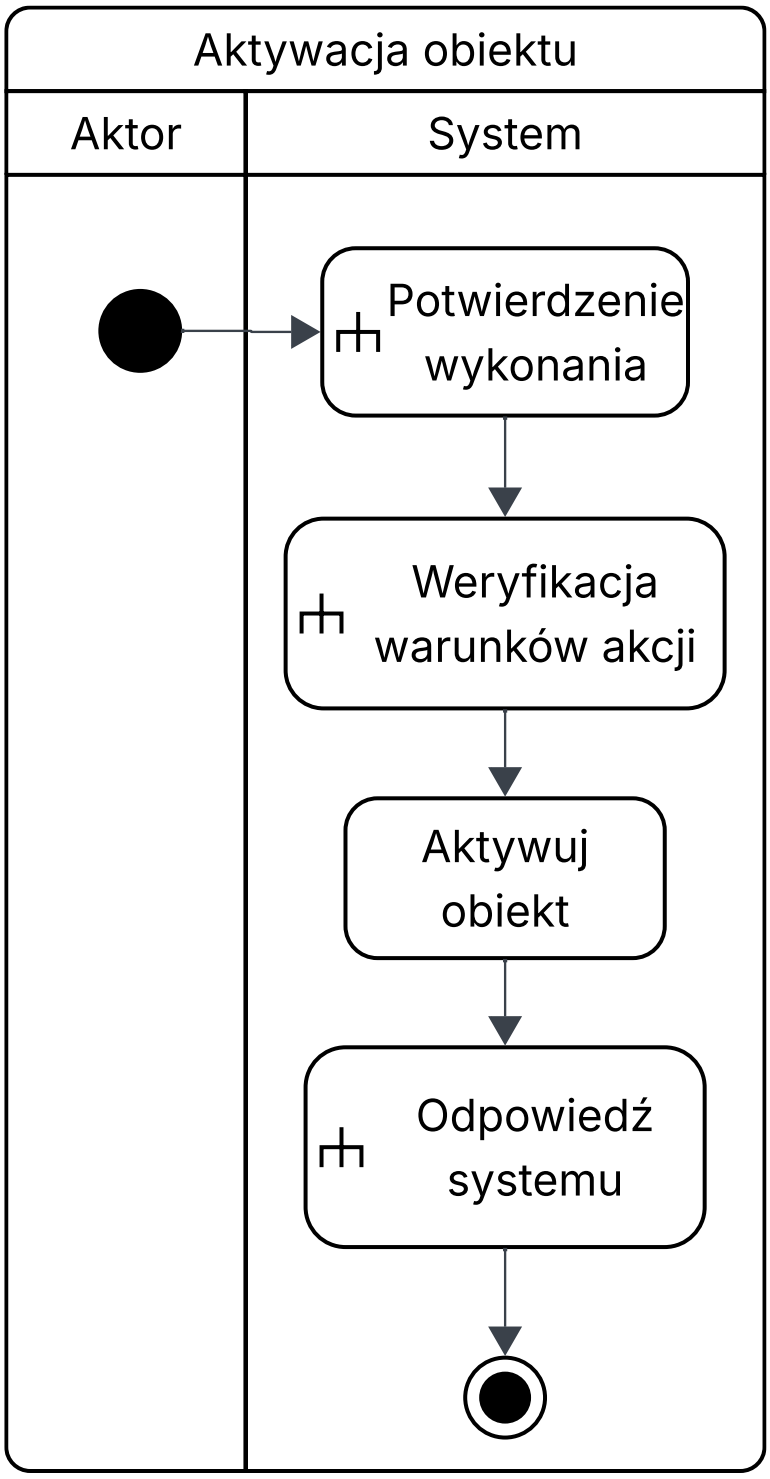
Rysunek 38 Diagram dodania obiektu. Źródło: opracowanie własne.

Zaprezentowany na *Rysunek 39* diagram usunięcia obiektu przedstawia operację trwałego usunięcia wybranego elementu z systemu Wellhicle. Przed przeprowadzeniem tej czynności aktor musi potwierdzić swój zamiar, aby uniknąć nieodwracalnych działań. Następnie system weryfikuje warunki wykonania akcji, ponieważ usunięcie może być ograniczone przez stan obiektu lub jego powiązania z innymi danymi. Jeśli wszystkie warunki są spełnione, system bezpiecznie usuwa obiekt. Po spełnieniu warunków system usuwa obiekt i zwraca odpowiedź potwierdzającą wykonanie operacji.



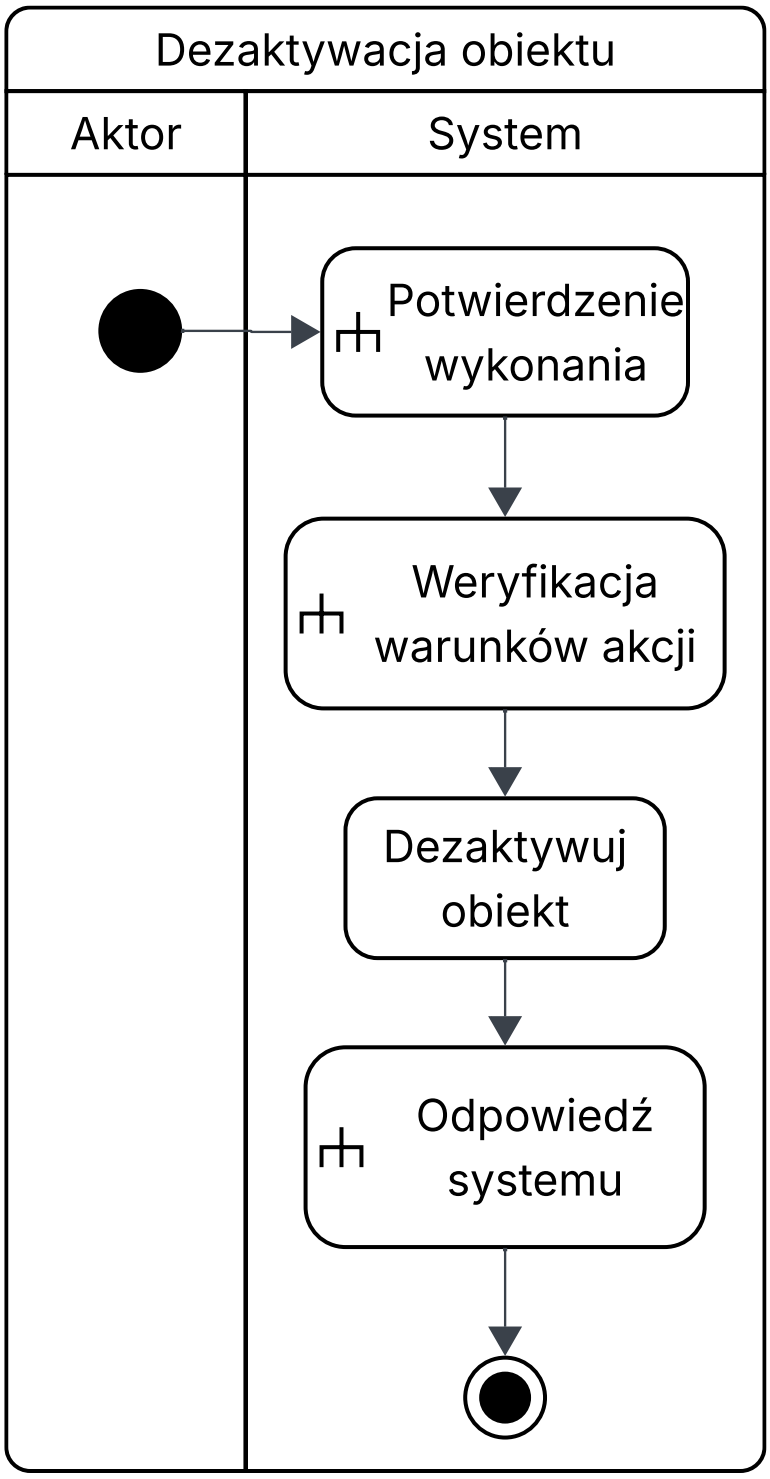
Rysunek 39 Diagram usunięcia obiektu. Źródło: opracowanie własne.

Diagram aktywacji obiektu przedstawiony na *Rysunek 40*, wykazuje podobny ogólny przebieg do diagramu usunięcia obiektu. Aktor inicjuje operację, po czym wymagane jest potwierdzenie jej realizacji oraz sprawdzenie warunków systemowych. Kluczowa różnica polega na skutku: aktywacja – jeśli spełnione są wymagane kryteria – zmienia status obiektu na aktywny, przywracając go tym samym do użytku.



Rysunek 40 Diagram aktywacji obiektu. Źródło: opracowanie własne.

Diagram dezaktywacji obiektu przedstawiony na *Rysunek 41*, działa analogicznie do diagramu aktywacji, jednak prowadzi do przeciwnego skutku. Aktor inicjuje operację, po czym wymagane jest potwierdzenie jej realizacji oraz sprawdzenie warunków systemowych. Jeżeli spełnione są określone kryteria, status obiektu zostaje zmieniony na nieaktywny. Oznacza to wyłączenie go z bieżącego użytkowania bez trwałego usunięcia z systemu.

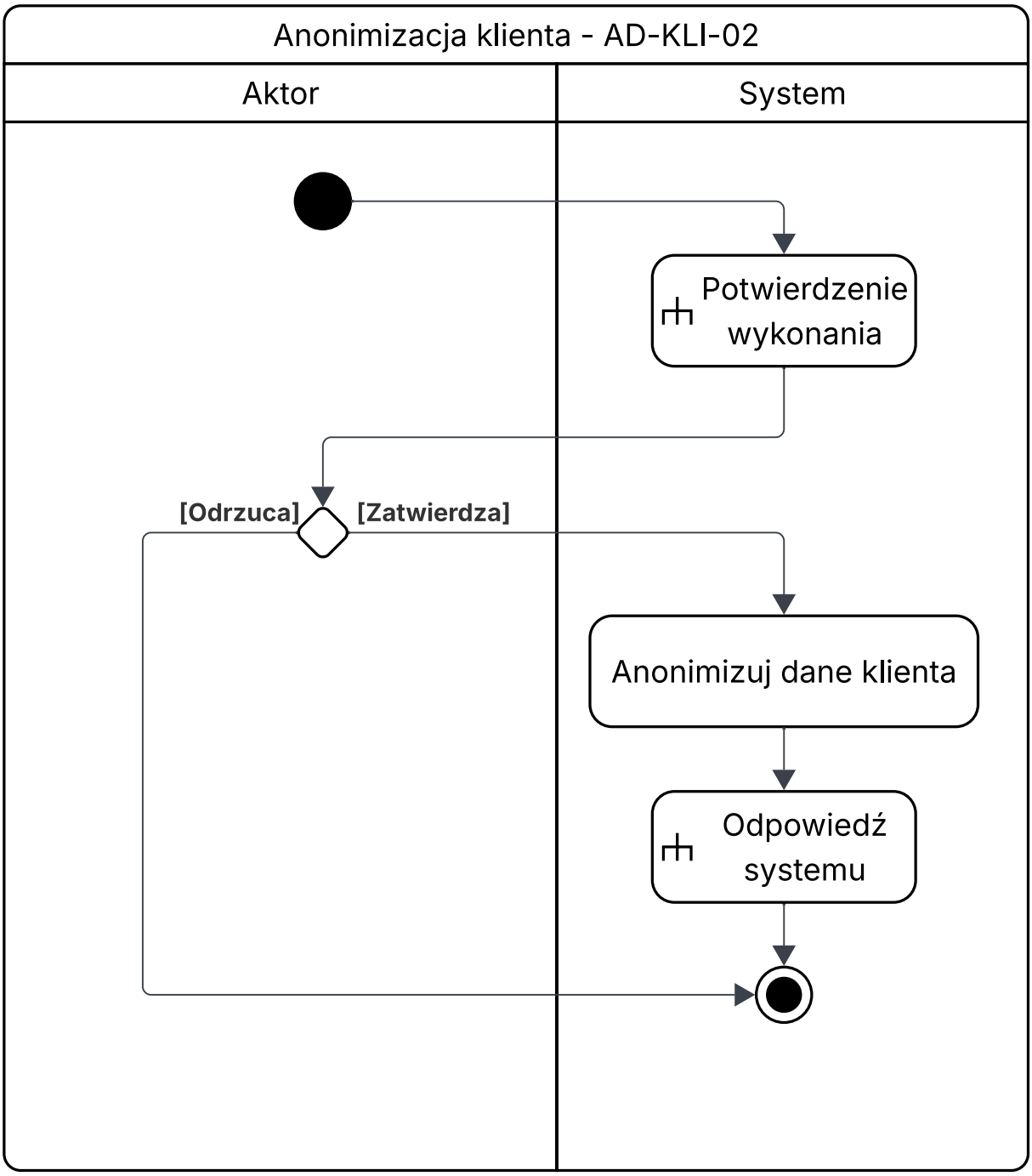


Rysunek 41 Diagram dezaktywacji obiektu. Źródło: opracowanie własne.

5.6.2. Anonimizacja danych klienta wypożyczalni

Rysunek 42 przedstawia diagram aktywności dla przypadku użycia anonimizacji danych klienta, stanowiący graficzną reprezentację wymagania funkcjonalnego WF-KLI-10 wraz z powiązanymi regułami biznesowymi, szerzej opisanymi w podrozdziale *5.3.3.2*. Operacja prowadzi do nieodwracalnego zastąpienia danych osobowych klienta wypożyczalni danymi neutralnymi. Czynność może zostać wykonana przez właściciela wypożyczalni oraz po uprzednim dezaktywowaniu profilu

klienta w systemie. Pełny proces wykonania anonimizacji opisano w *Tabela 52* zawierającej scenariusz przypadku użycia UC-KLI-02.



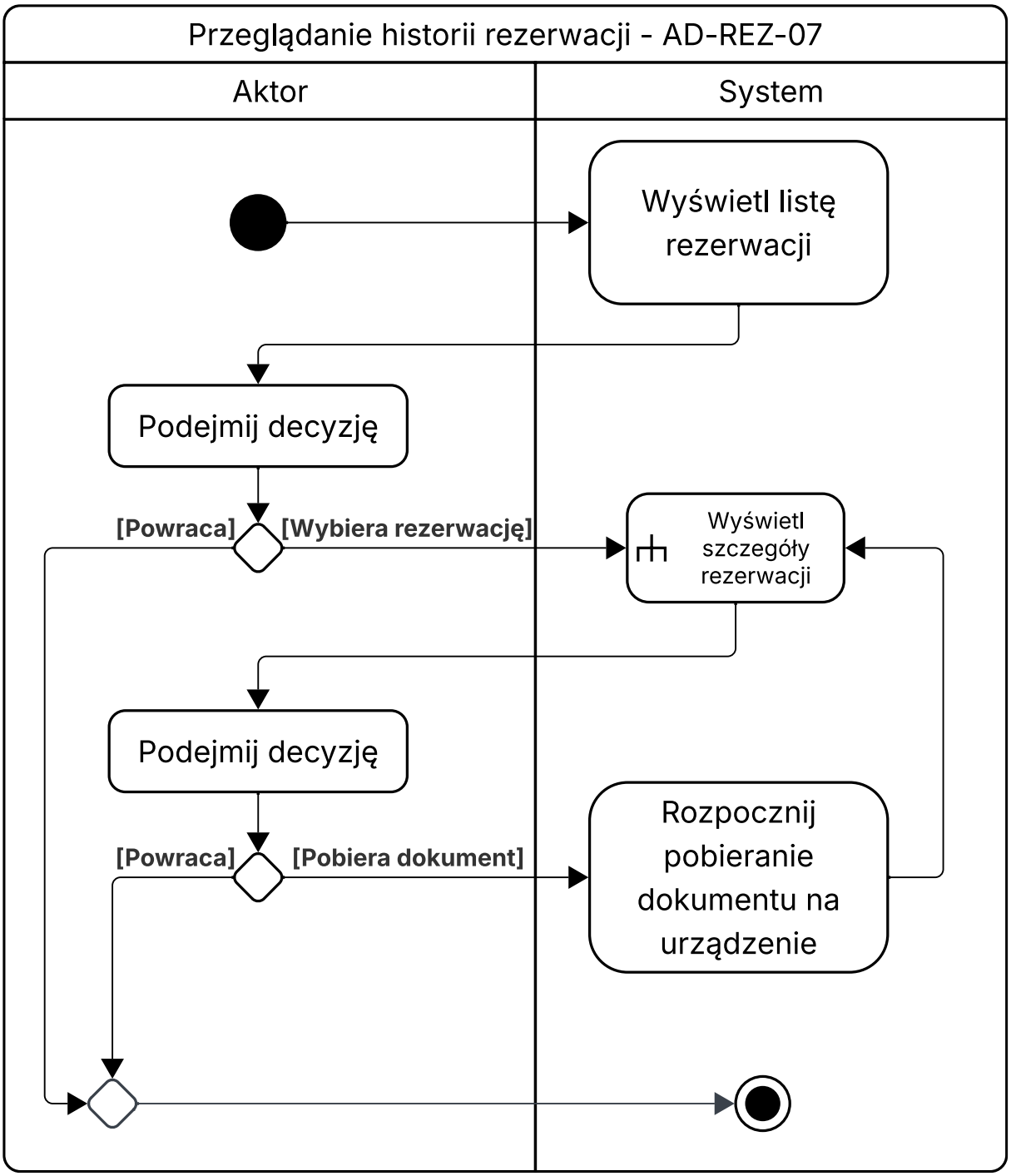
Rysunek 42 Diagram aktywności – anonimizacja klienta AD-KLI-02. Źródło: opracowanie własne.

Tabela 52 Scenariusz przypadku użycia – anonimizacja klienta. Źródło: opracowanie własne.

Nazwa przypadku użycia	Anonimizacja klienta	**Id.**
		UC-KLI-02
Aktor	Właściciel wypożyczalni.	
Cel	Nieodwracalnie zanonimizowanie danych osobowych klienta.	
Warunki wstępne	Brak niezakończonych rezerwacji klienta. Aktor posiada dostęp do przeprowadzenia anonimizacji. Profil klienta został dezaktywowany.	
Efekt Końcowy	Dane osobowe klienta zastąpione danymi neutralnymi.	
Scenariusz główny		
Krok	**Akcja**	
1.	Aktor uruchamia akcję anonimizacji klienta.	
2.	System wyświetla komunikat potwierdzenia operacji.	
3.	Aktor zatwierdza komunikat.	
4.	System anonimizuje dane klienta.	
5.	System wyświetla komunikat o pomyślnie wykonanej operacji.	
6.	Aktor zatwierdza komunikat -> przejście do [UC-KLI-01].	
Scenariusze alternatywne		
3a.	Aktor odrzuca komunikat -> przejście do [UC-KLI-01].	
6a.	Błąd systemu: zmiany nie zostają wprowadzone, system wyświetla komunikat o błędzie -> przejście do kroku 7.	

5.6.3. Przeglądanie historii rezerwacji przez klienta

Rysunek 43 przedstawia diagram aktywności dla przypadku użycia przeglądania historii rezerwacji, stanowiący reprezentację wymagania funkcjonalnego WF-SWY-09 wraz z powiązanymi regułami biznesowymi, szczegółowo opisanymi w podrozdziale *5.3.9.2.* Operacja umożliwia klientowi wypożyczalni przeglądanie listy swoich rezerwacji wraz z dostępem do ich szczegółów oraz możliwością pobrania powiązanych dokumentów. Pełny proces wykonania operacji opisano w *Tabela 53* zawierającej scenariusz przypadku użycia UC-REZ-07.



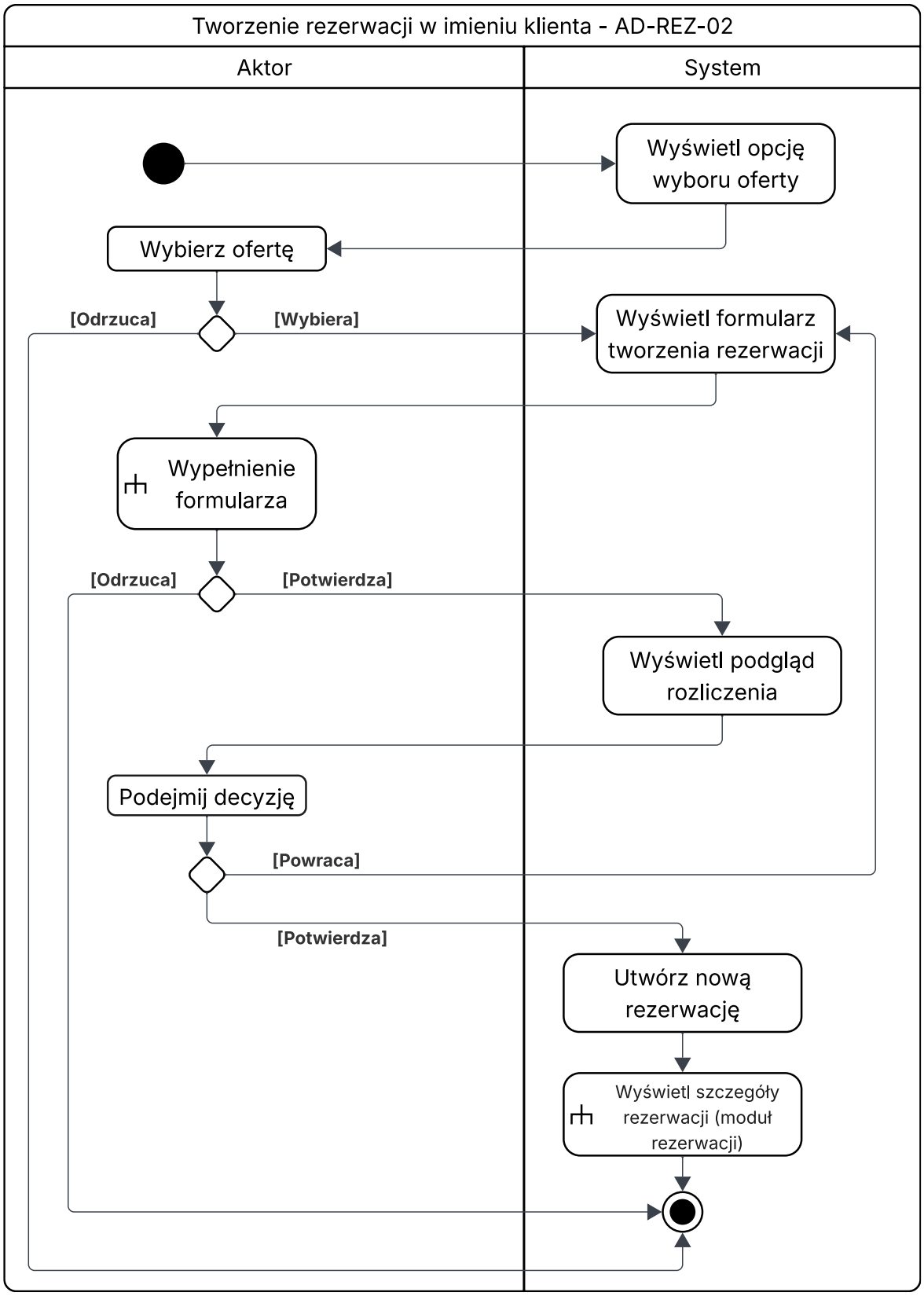
Rysunek 43 Diagram aktywności – przeglądanie historii rezerwacji AD-REZ-07. Źródło: opracowanie własne.

Tabela 53 Scenariusz przypadku użycia – przeglądanie historii rezerwacji. Źródło: opracowanie własne.

Nazwa przypadku użycia	Przeglądanie historii rezerwacji	**Id.**
		UC-REZ-07
Aktor	Klient wypożyczalni	
Cel	Przegląd listy rezerwacji, podgląd szczegółów oraz uruchamianie operacji jednostkowych.	
Warunki wstępne	Aktor posiada uprawnienia do przeglądania swoich rezerwacji.	
Efekt Końcowy	Wyświetlona lista rezerwacji z opcją filtrowania oraz możliwość wykonania operacji pobrania dokumentów.	
Scenariusz główny		
Krok	**Akcja**	
1.	Aktor otwiera panel rezerwacji.	
2.	System wyświetla listę rezerwacji, do których aktor ma dostęp z możliwością filtrowania. Dla każdej rezerwacji dostępna jest opcja wejścia w szczegóły.	
3.	Aktor wybiera rezerwację -> przejście do szczegółów rezerwacji z możliwością pobrania dokumentów.	
4.	Aktor wybiera opcję pobrania dokumentów.	
5.	System rozpoczyna pobieranie dokumentów na urządzenie.	
Scenariusze alternatywne		
3a.	Aktor wychodzi z panelu rezerwacji -> koniec przypadku użycia.	
4a.	Aktor filtruje listę rezerwacji -> przejście do kroku 2.	
4b.	Aktor wychodzi ze szczegółów rezerwacji -> przejście do kroku 2.	

5.6.4. Tworzenie rezerwacji w imieniu klienta

Rysunek 44 przedstawia diagram aktywności dla przypadku użycia tworzenia rezerwacji w imieniu klienta, odpowiadającego wymaganiu funkcjonalnemu WF-REZ-03, którego pełna specyfikacja wraz z regułami biznesowymi została zawarta w podrozdziale *5.3.5.2*. Operacja umożliwia pracownikowi biurowemu lub właścicielowi wypożyczalni dodanie nowej rezerwacji, poprzez wieloetapowy proces wyboru oferty, wypełnienia formularza rezerwacji oraz zatwierdzenia rozliczenia. Pełny przebieg procesu wraz ze scenariuszami alternatywnymi opisano w *Tabela 54* zawierającej scenariusz przypadku użycia UC-REZ-02.



Rysunek 44 Diagram aktywności – tworzenie rezerwacji w imieniu klienta AD-REZ-02. Źródło: opracowanie własne.

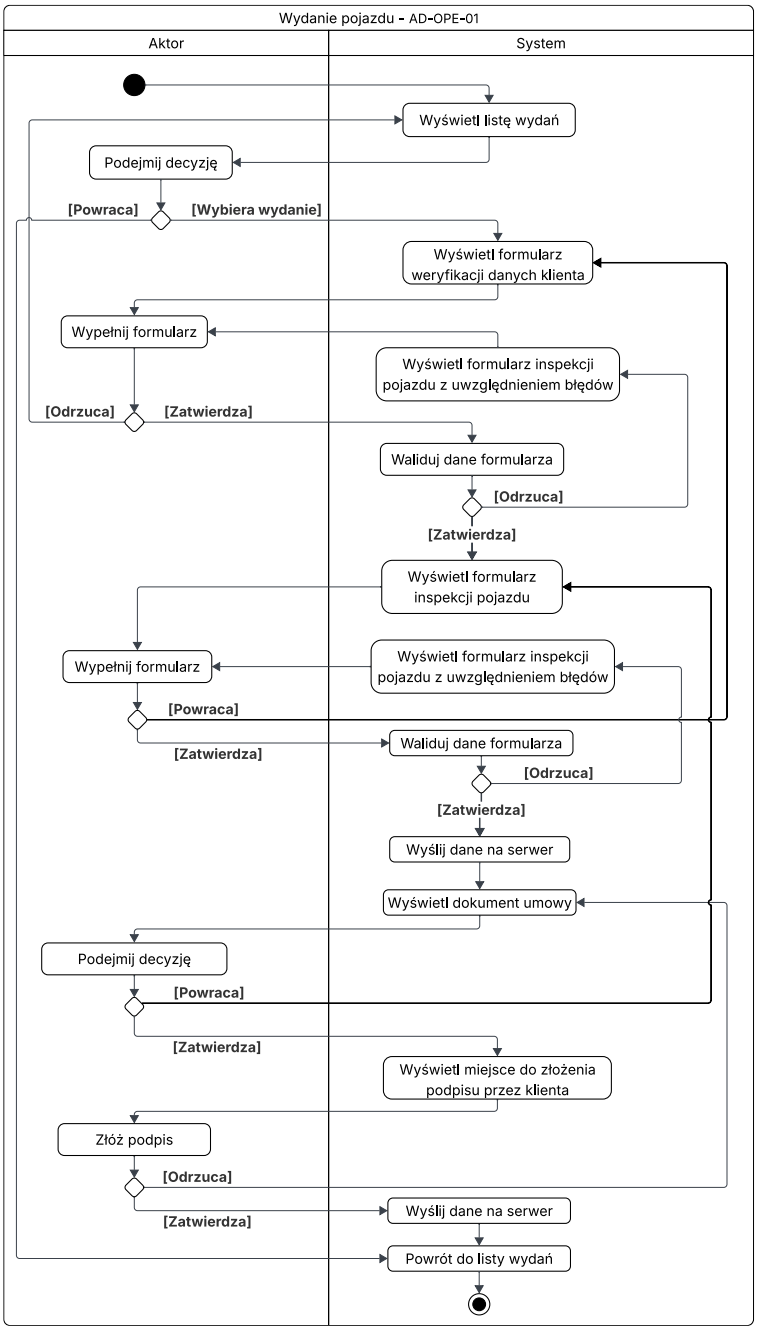
Tabela 54 Scenariusz przypadku użycia – tworzenie rezerwacji w imieniu klienta. Źródło: opracowanie własne.

Nazwa przypadku użycia	Utworzenie rezerwacji w imieniu klienta.	**Id.**
		UC-REZ-02
Aktor	Pracownik biurowy, właściciel wypożyczalni	
Cel	Dodanie nowej rezerwacji pojazdu.	
Warunki wstępne	Aktor ma dostęp do dodawania rezerwacji.	
Efekt Końcowy	Dodana nowa rezerwacja.	
Scenariusz główny		
Krok	**Akcja**	
1.	Aktor uruchamia akcję dodania rezerwacji.	
2.	System wyświetla formularz wyboru oferty.	
3.	Aktor wybiera ofertę.	
4.	System wyświetla formularz konfiguracji rezerwacji.	
5.	Aktor wypełnia i zatwierdza formularz.	
6.	System waliduje formularz.	
7.	System wyświetla formularz podglądu rozliczenia	
8.	Aktor zatwierdza formularz	
9.	System dodaje rezerwację do bazy danych.	
10.	System wyświetla komunikat o pomyślnym dodaniu rezerwacji.	
11.	Aktor potwierdza komunikat -> przejście do [UC-REZ-01].	
Scenariusze alternatywne		
3a.	Aktor odrzuca formularz -> przejście do [UC-REZ-01].	
5a.	Aktor odrzuca formularz -> przejście do [UC-REZ-01].	
6a.	Błąd walidacji danych: system wyświetla formularz dodawania rezerwacji z błędami -> przejście do kroku 5.	
8a.	Aktor powraca do formularza dodawania rezerwacji -> przejście do kroku 4.	
8b.	Aktor odrzuca formularz -> przejście do [UC-REZ-01].	
10a.	Błąd systemu: zmiany nie zostają wprowadzone, system wyświetla komunikat o błędzie -> przejście do kroku 9.	

5.6.5. Wydanie pojazdu

Diagram aktywności, przedstawiony na *Rysunek 45*, ilustruje proces wydania pojazdu klientowi realizowany przez pracownika operacyjnego z poziomu aplikacji mobilnej. Szczegółowy opis funkcjonalności modułu operacyjnego wraz z powiązanymi regułami biznesowymi zawarto w

podrozdziale *5.3.8.2.* Operacja obejmuje wieloetapowy proces weryfikacji danych klienta, przeprowadzenia inspekcji pojazdu oraz podpisania dokumentu umowy. Pełny przebieg procesu wraz ze scenariuszami alternatywnymi opisano w *Tabela 55* ze scenariuszem przypadku użycia UC-OPE-01.



Rysunek 45 Diagram aktywności – wydanie pojazdu AD-OPE-01. Źródło: opracowanie własne.

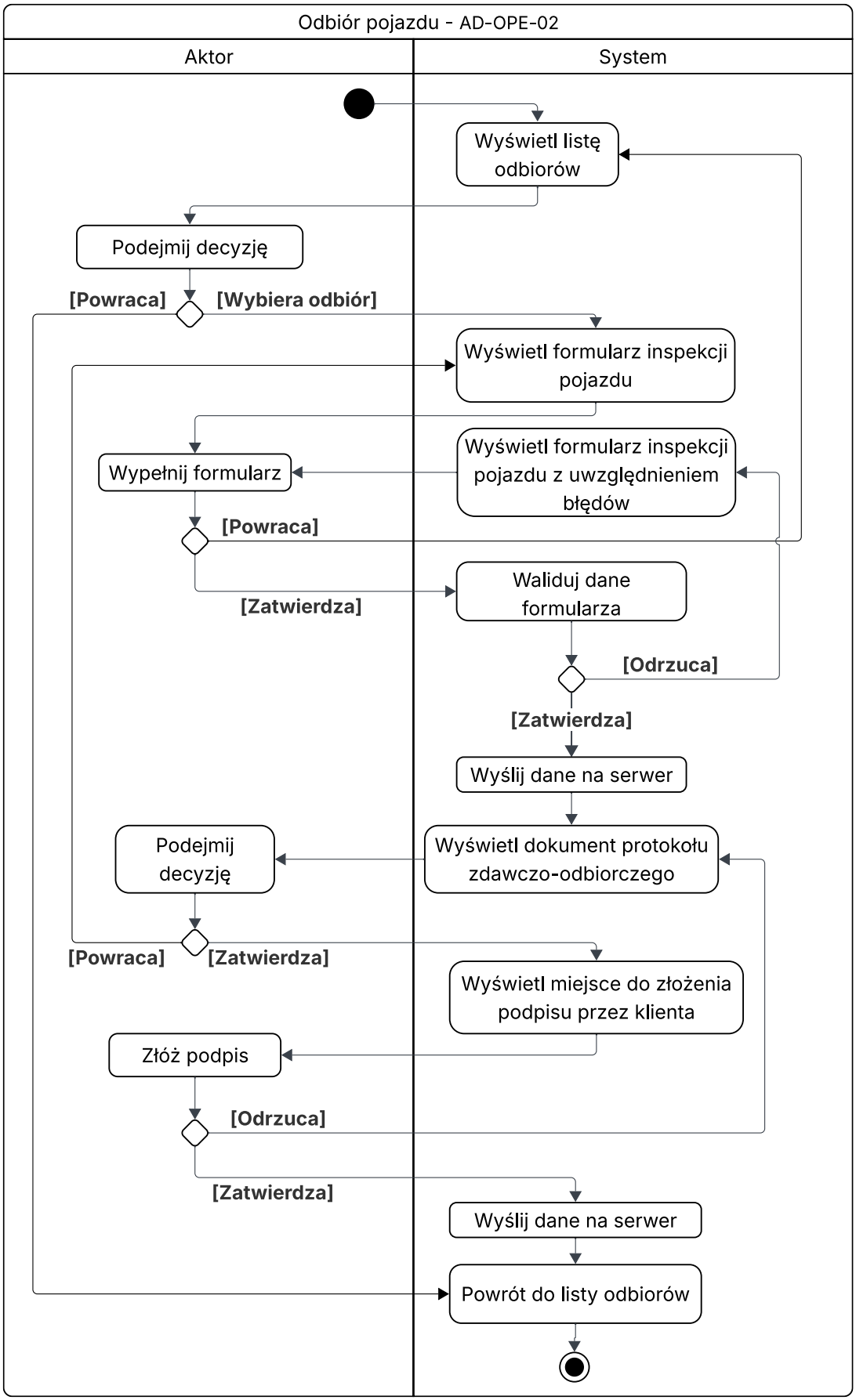
Tabela 55 Scenariusz przypadku użycia – wydanie pojazdu. Źródło: opracowanie własne.

Nazwa przypadku użycia	Wydanie pojazdu	**Id.**
		UC-OPE-01
Aktor	Pracownik operacyjny.	
Cel	Wydanie pojazdu klientowi zgodnie z przewidzianą rezerwacją.	
Warunki wstępne	Rezerwacja jest potwierdzona. Pojazd jest możliwy do wydania. Aktor ma dostęp do panelu wydań i konkretnego wydania.	
Efekt Końcowy	Pojazd wydany klientowi.	
Scenariusz główny		
Krok	**Akcja**	
1.	Aktor wchodzi do panelu wydań	
2.	System wyświetla listę planowanych wydań.	
3.	Aktor wybiera wydanie.	
4.	System wyświetla formularz weryfikacji danych klienta.	
5.	Aktor wypełnia i zatwierdza formularz.	
6.	System wyświetla formularz inspekcji pojazdu.	
7.	Aktor wypełnia i zatwierdza formularz.	
8.	System waliduje wprowadzone do formularza dane.	
9.	System wysyła dane na serwer.	
10.	System wyświetla dokument umowy.	
11.	Aktor zatwierdza dokument.	
12.	System wyświetla miejsce do złożenia podpisu przez klienta.	
13.	Klient składa i zatwierdza podpis.	
14.	System wysyła dane na serwer i kończy operację -> powrót do kroku 2.	
Scenariusze alternatywne		
2a.	Jeżeli lista planowanych wydań jest pusta -> system wyświetla informacje o braku planowanych wydań.	
3a.	Aktor wychodzi z panelu wydań -> koniec przypadku użycia.	
4a.	Aktor wybiera opcję powrotu -> przejście do kroku 2.	
5a.	Aktor odrzuca dane klienta-> przejście do kroku 4.	
5b.	Aktor wybiera opcję powrotu -> przejście do kroku 2.	
7a.	Aktor odrzuca formularz -> przejście do kroku 4.	
7b.	Aktor wybiera opcję powrotu -> przejście do kroku 4.	
8a.	Błąd walidacji danych, system zwraca formularz z uwzględnieniem błędów -> powrót do kroku 6.	

10a.	Błąd przesłania danych: system wyświetla komunikat -> przejście do kroku 6.
11a.	Aktor wybiera opcję powrotu -> przejście do kroku 6.
11b.	Aktor odrzuca dokument -> przejście do kroku 6.
13a.	Aktor odrzuca złożenie podpisu -> przejście do kroku 10.

5.6.6. Odbiór pojazdu

Przypadek użycia, zilustrowany na *Rysunek 46*, opisuje proces odbioru pojazdu od klienta i zwrócenia go na stan wypożyczalni, stanowiący kolejny z przypadków użycia modułu operacyjnego, którego pełna specyfikacja funkcjonalności opisana została w podrozdziale *5.3.8.2.* Podobnie jak w przypadku wydania, operacja prowadzona jest przez pracownika operacyjnego z poziomu aplikacji mobilnej i obejmuje przeprowadzenie inspekcji pojazdu oraz podpisanie protokołu zdawczo-odbiorczego. Scenariusz przypadku użycia wraz z pełnym przebiegiem procesu opisano w *Tabela 56*.



Rysunek 46 Diagram aktywności – odbiór pojazdu AD-OPE-02. Źródło: opracowanie własne.

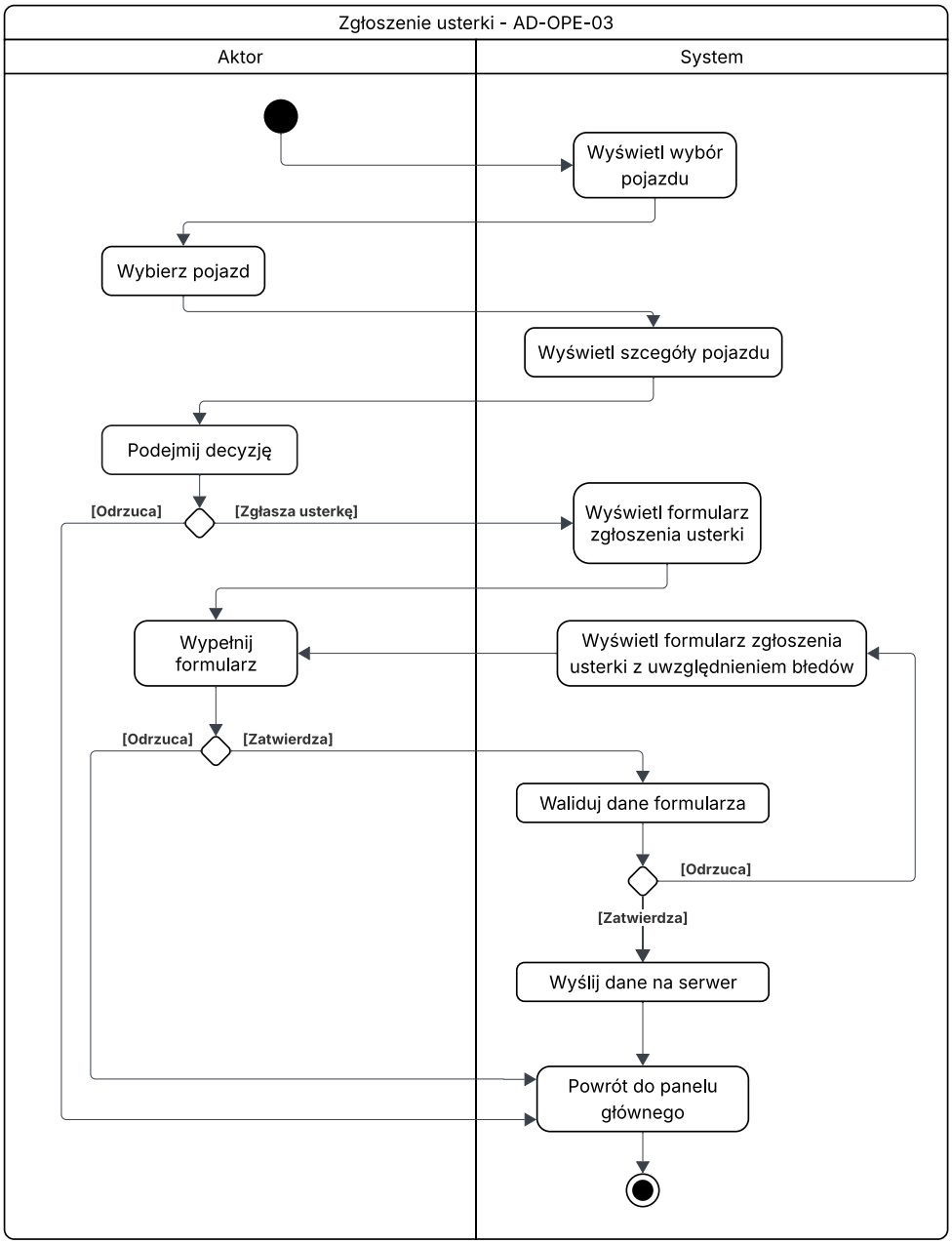
Tabela 56 Scenariusz przypadków użycia – odbiór pojazdu. Źródło: opracowanie własne.

Nazwa przypadku użycia	Odbiór pojazdu.	**Id.**
		UC-OPE-02
Aktor	Pracownik operacyjny.	
Cel	Zakończenie najmu pojazdu poprzez odbiór od klienta i zwrócenie na stan wypożyczalni. Aktor ma dostęp do panelu odbiorów i konkretnego odbioru.	
Warunki wstępne	Obecna data jest w okresie rezerwacji.	
Efekt Końcowy	Pojazd zwrócony na stan wypożyczalni.	
Scenariusz główny		
Krok	**Akcja**	
1	Aktor wchodzi do panelu odbiorów.	
2.	System wyświetla listę planowanych odbiorów z możliwością filtrowania.	
3.	Aktor wybiera odbiór.	
4.	System wyświetla szczegóły odbioru.	
5.	Aktor rozpoczyna odbiór pojazdu.	
6.	System wyświetla formularz inspekcji pojazdu.	
7.	Aktor wypełnia i zatwierdza formularz.	
8.	System waliduje dane formularza.	
9.	System wysyła dane na serwer.	
10.	System wyświetla dokument protokołu zdawczo-odbiorczego.	
11.	Aktor zatwierdza dokument.	
12.	System wyświetla miejsce do złożenia podpisu przez klienta.	
13.	Klient składa podpis i go zatwierdza.	
14.	System wysyła dane na serwer i kończy operację -> powrót do kroku 2.	
Scenariusze alternatywne		
2a.	Jeżeli lista planowanych wydań jest pusta -> system wyświetla komunikat o braku planowanych wydań.	
3a.	Aktor wychodzi z panelu odbiorów-> koniec przypadku użycia.	
5a.	Aktor wybiera opcję powrotu -> przejście do kroku 2.	
7a.	Aktor odrzuca formularz -> przejście do kroku 4.	
8a.	Błąd walidacji danych, system zwraca formularz z uwzględnieniem błędów -> powrót do kroku 6.	
10a.	Błąd przesłania danych: system wyświetla komunikat -> przejście do kroku 6.	
11a.	Aktor odrzuca dokument -> przejście do kroku 6.	

13a.	Aktor odrzuca złożenie podpisu -> przejście do kroku 10.
14a.	Błąd przesłania danych: system wyświetla komunikat -> przejście do kroku 10.

5.6.7. Zgłoszenie usterki

Na *Rysunek 47* przedstawiono diagram aktywności dla przypadku użycia UC-OPE-03, odpowiadającego wymaganiu funkcjonalnemu WF-OPE-11, którego pełna specyfikacja wraz z regułami biznesowymi została zawarta w podpunkcie *5.3.8.2.* Operacja realizowana jest przez pracownika operacyjnego z poziomu aplikacji mobilnej. Pełny scenariusz przypadku użycia opisano w *Tabela 57*.



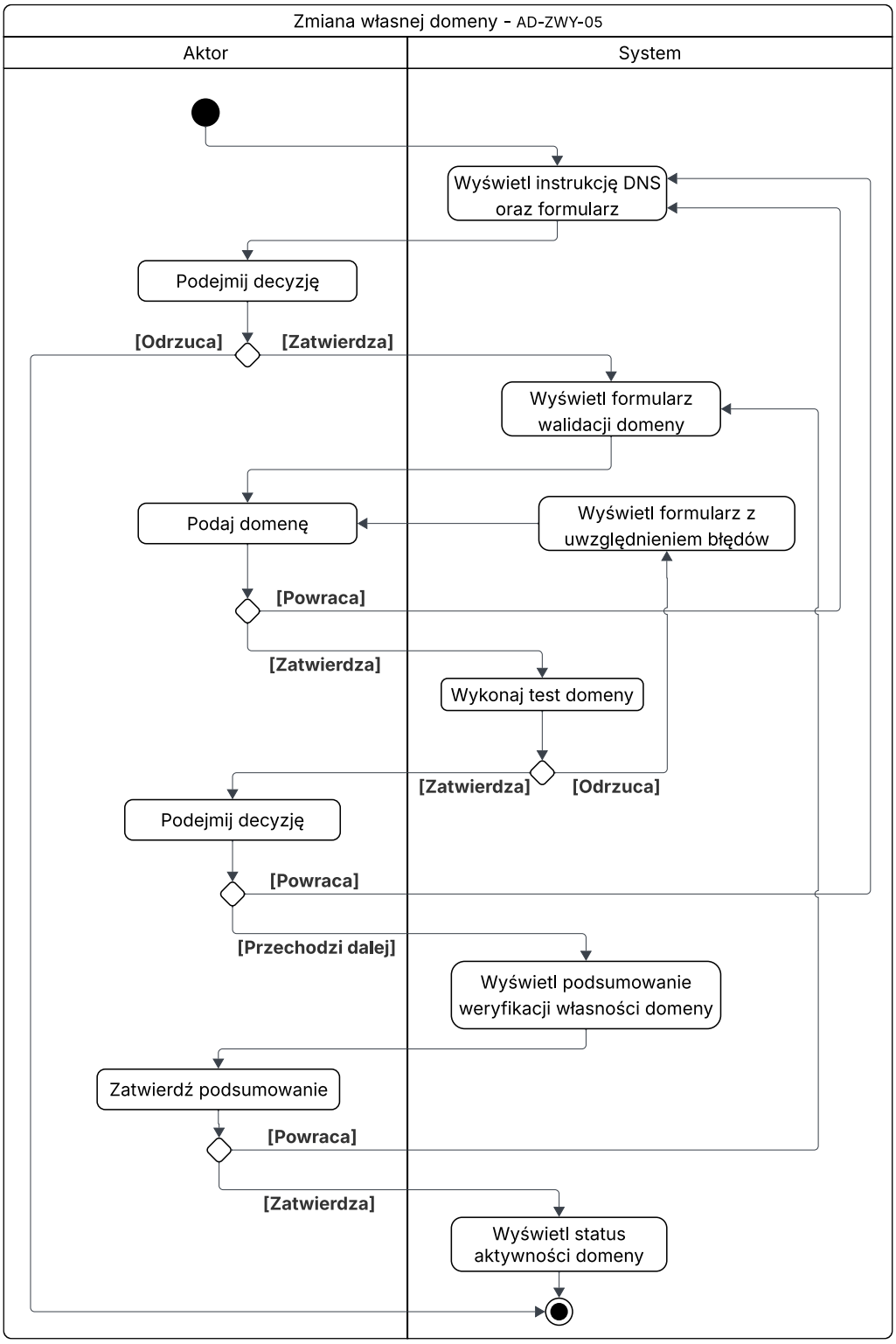
Rysunek 47 Diagram aktywności – zgłoszenie usterki AD-OPE-03. Źródło: opracowanie własne.

Tabela 57 Scenariusz przypadku użycia – zgłoszenie usterki. Źródło: opracowanie własne.

Nazwa przypadku użycia	Zgłoszenie usterki.	**Id.**
		UC-OPE-03
Aktor	Pracownik operacyjny.	
Cel	Zarejestrowanie wykrytej usterki pojazdu.	
Warunki wstępne	Aktor ma dostęp do panelu zgłaszania usterek.	
Efekt Końcowy	Zgłoszona usterka pojazdu.	
Scenariusz główny		
Krok	**Akcja**	
1.	Aktor wchodzi do panelu zgłaszania usterek.	
2.	System wyświetla listę pojazdów z możliwością filtrowania.	
3.	Aktor wybiera pojazd.	
4.	System wyświetla szczegóły pojazdu.	
5.	Aktor wybiera opcję zgłoszenia usterki.	
6.	System wyświetla formularz zgłoszenia usterki.	
7.	Aktor wypełnia formularz i go zatwierdza.	
8.	System waliduje wprowadzone dane.	
9.	System wysyła dane na serwer i zwraca komunikat o powodzeniu operacji.	
10.	Aktor zatwierdza komunikat.	
11.	System przekierowuje do panelu głównego -> koniec przypadku użycia.	
Scenariusze alternatywne		
2a.	Aktor filtruje listę wydań -> przejście do kroku 2.	
3a.	Aktor wychodzi z panelu zgłaszania usterek -> koniec przypadku użycia.	
5a.	Aktor wybiera opcję powrotu ->przejście do kroku 2.	
7a.	Aktor odrzuca formularz -> przejście do kroku 2.	
8a.	Błąd walidacji danych: system zwraca formularz z uwzględnieniem błędów -> przejście do kroku 7.	
10a.	Błąd przesłania danych: system wyświetla komunikat -> przejście do kroku 7.	

5.6.8. Zmiana własnej domeny

Rysunek 48 przedstawia diagram aktywności dla przypadku użycia zmiany własnej domeny, odpowiadającego wymaganiu funkcjonalnemu WF-ZWY-10, którego pełna specyfikacja została zawarta w podpunkcie *5.3.1.2.* Operacja umożliwia właścicielowi wypożyczalni podłączenie własnej domeny poprzez wieloetapowy formularz konfiguracji. Szczegółowy scenariusz przypadku użycia UC-ZWY-05 wraz z przebiegiem alternatywnym opisano w *Tabela 58*.



Rysunek 48 Diagram aktywności – zmiana własnej domeny AD-ZWY-05. Źródło: opracowanie własne.

Tabela 58 Scenariusz przypadku użycia – zmiana własnej domeny. Źródło: opracowanie własne.

Nazwa przypadku użycia	Zmiana własnej domeny	**Id.** UC-ZWY-05
Aktor	Właściciel wypożyczalni	

Cel	Podłączenie własnej domeny.
Warunki wstępne	Aktor ma dostęp do zmiany domeny.
Efekt Końcowy	Zaktualizowana domena.
Scenariusz główny	
Krok	**Akcja**
1.	Aktor uruchamia akcję zmiany własnej domeny.
2.	System wyświetla instrukcję DNS oraz formularz zmiany domeny.
3.	Aktor zatwierdza formularz.
4.	System wyświetla formularz walidacji domeny.
5.	Aktor podaje swoją domenę.
6.	System wykonuje test domeny.
7.	System wyświetla opcję przejścia do następnego kroku.
8.	Aktor wybiera opcję przejścia dalej.
9.	System wyświetla podsumowanie weryfikacji własności domeny.
10.	Aktor zatwierdza okno podsumowania
11.	System wyświetla status aktywności domeny -> koniec przypadku użycia.
Scenariusze alternatywne	
3a.	Aktor odrzuca formularz -> koniec przypadku użycia.
7a.	Błąd walidacji danych: system wyświetla formularz walidacji domeny z błędami -> powrót do kroku 5.
7b.	Aktor wybiera opcję powrotu -> powrót do kroku 2.
10a.	Aktor wybiera opcję powrotu -> powrót do kroku 4.

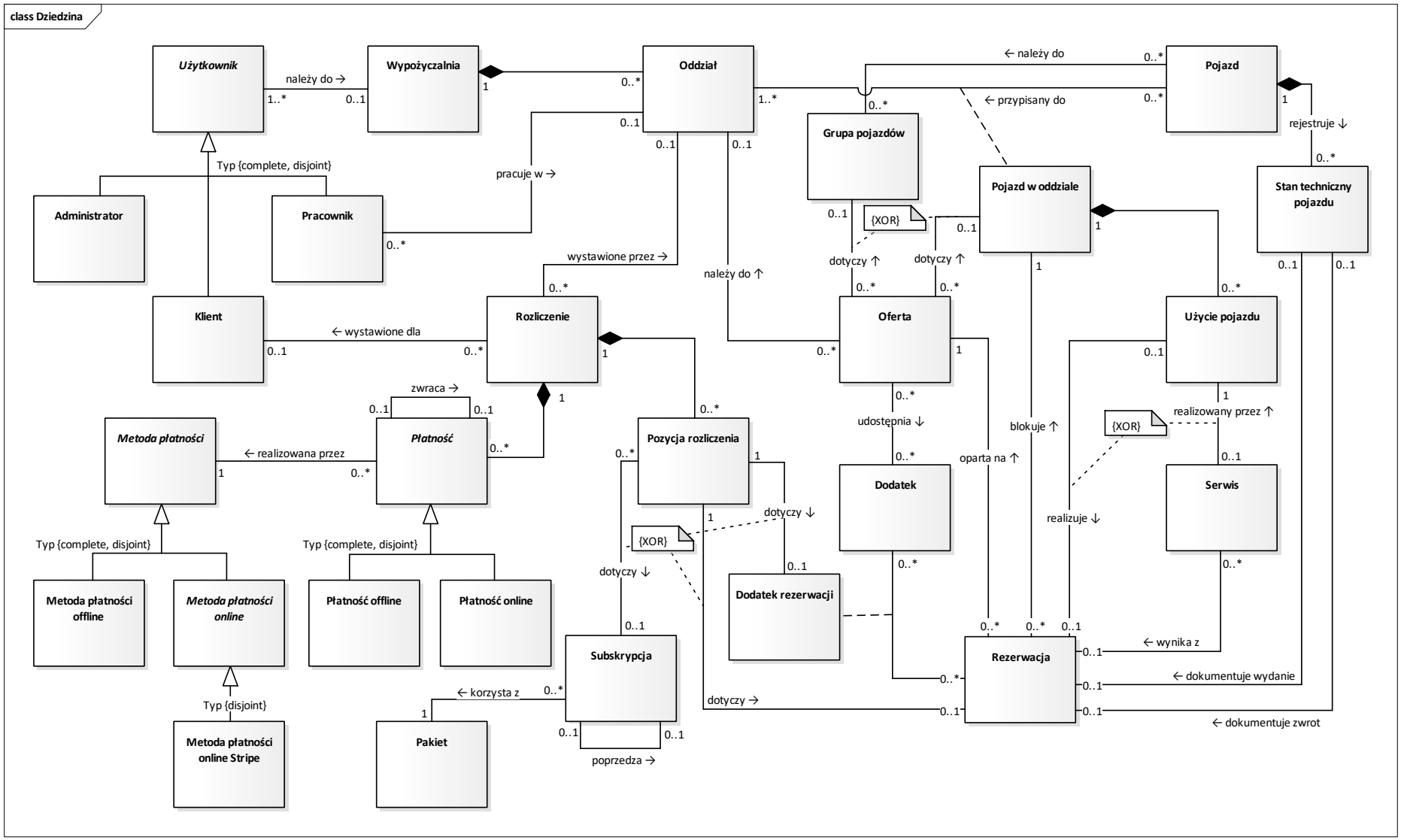
5.7. Analityczny diagram klas

Dziedzinę systemu Wellhicle ujęto w postaci analitycznego modelu klas (ang. Analytical Class Diagram, dalej ACD). Tam, gdzie przypadki użycia i diagramy aktywności opisują dziedzinę od strony zachowań, model klas ujmuje ją statycznie: wskazuje pojęcia tworzące dziedzinę, ich istotne cechy oraz powiązania między nimi i scala je w jeden spójny obraz strukturalny.

Nazwy klas, atrybutów i ról zapisano po polsku, zgodnie z językiem dziedziny, oraz bez wskazywania typów technicznych. Dzięki temu diagram koncentruje się na tym, co system ma reprezentować, a nie na tym, jak zostanie to zapisane w kodzie. Szczegóły implementacyjne tej samej dziedziny przedstawia projektowy model klas opisany w podrozdziale 6.3.

Całość dziedziny ujmuje jeden diagram, przedstawiający wszystkie klasy i ich powiązania bez atrybutów na jednym obrazie (Rysunek 49). Porządkują go dwie warstwy. Warstwa platformowa obejmuje wypożyczalnię jako instancję platformy wraz z oddziałami, pakietami i subskrypcjami, a rdzeń operacyjny grupuje użytkowników, flotę, oferty wraz z rezerwacjami i finanse. Centralnym

punktem jest wypożyczalnia, w której kontekście istnieją pozostałe byty i która łączy obie warstwy. Na styku obszarów uwidaczniają się klasy przekrojowe, wiążące użytkowników, flotę, oferty i finanse w jedną dziedzinę. Najważniejszą z nich jest pracownik, którego aktywności sięgają wielu obszarów: obsługuje flotę i serwis, realizuje wydanie oraz zwrot pojazdu w rezerwacji, a także wystawia rozliczenia. Podobnym łącznikiem perspektyw jest pozycja rozliczenia.

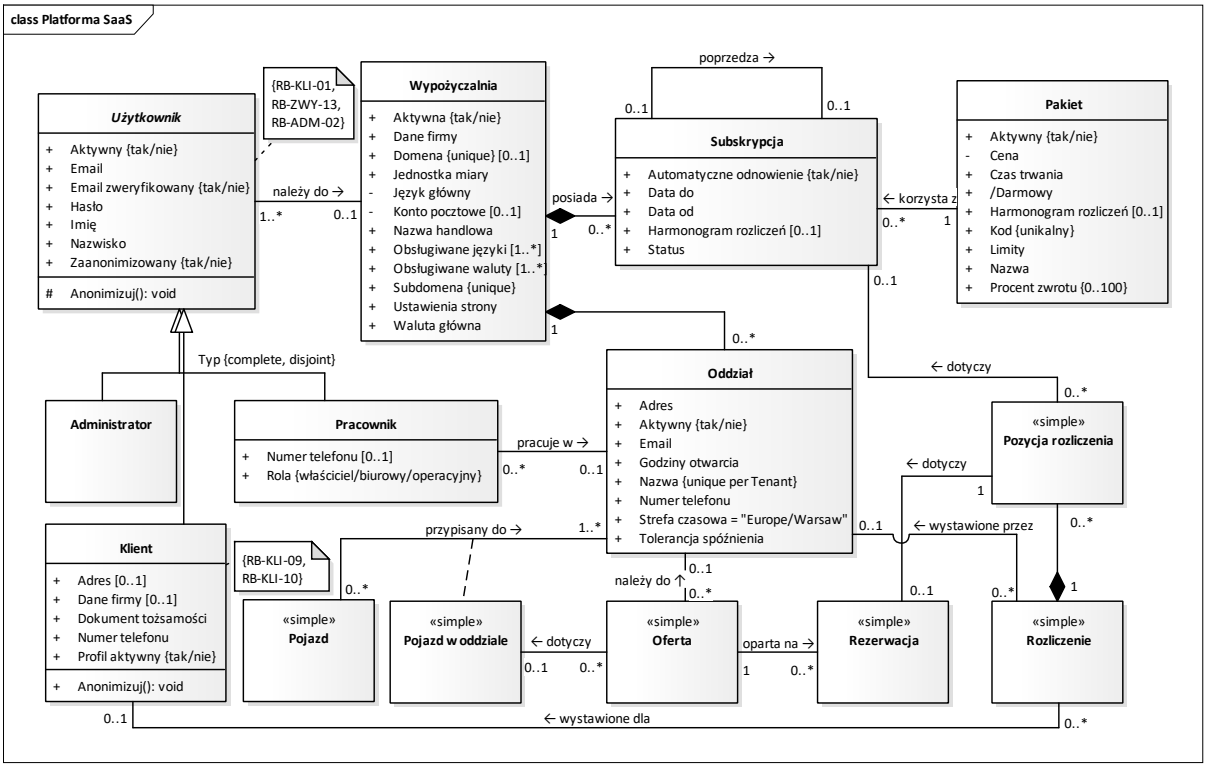


Rysunek 49 Analityczny diagram klas – model całościowy dziedziny (bez atrybutów). Źródło: opracowanie własne.

Ze względu na liczbę klas szczegółowy opis dziedziny prowadzimy w czterech perspektywach (punkty *5.7.1*, *5.7.2*, *5.7.3, 5.7.4*), dzielonych po granicach obszarów dziedziny. Wspomniane klasy przekrojowe powtarzają się w kilku perspektywach. W tej, w której klasa stanowi jedynie kontekst powiązania, a nie przedmiot opisu, pokazana jest w formie uproszczonej, bez atrybutów i powiązań wewnętrznych. Oznacza ją wprowadzony na potrzeby pracy stereotyp <<*simple*>>, a jej pełną definicję zawiera macierzysta perspektywa. Utrzymuje to czytelność granic perspektyw bez utraty widoczności powiązań między nimi.

5.7.1. Platforma SaaS

Perspektywa platformy SaaS przedstawia wypożyczalnię jako klienta platformy, wraz z jej strukturą organizacyjną (oddziałami, pakietami i subskrypcjami) oraz użytkownikami działającymi w jej kontekście. Jej strukturę ilustruje *Rysunek 50*.



Rysunek 50 Analityczny diagram klas – fragment: perspektywa platformy SaaS. Źródło: opracowanie własne.

Centralnym pojęciem jest *Wypożyczalnia*, odpowiednik tenanta: logicznie wydzielona, samodzielna instancja korzystająca ze wspólnej platformy zgodnie z modelem wielotenantowości. Jej tożsamość na platformie wyznacza unikalna subdomena, a konfiguracja, taka jak obsługiwane waluty i języki, określa zakres działania. Wypożyczalnia dzieli się na *Oddziały*, czyli jednostki obsługi pojazdów przypisane do dokładnie jednej wypożyczalni (kompozycja).

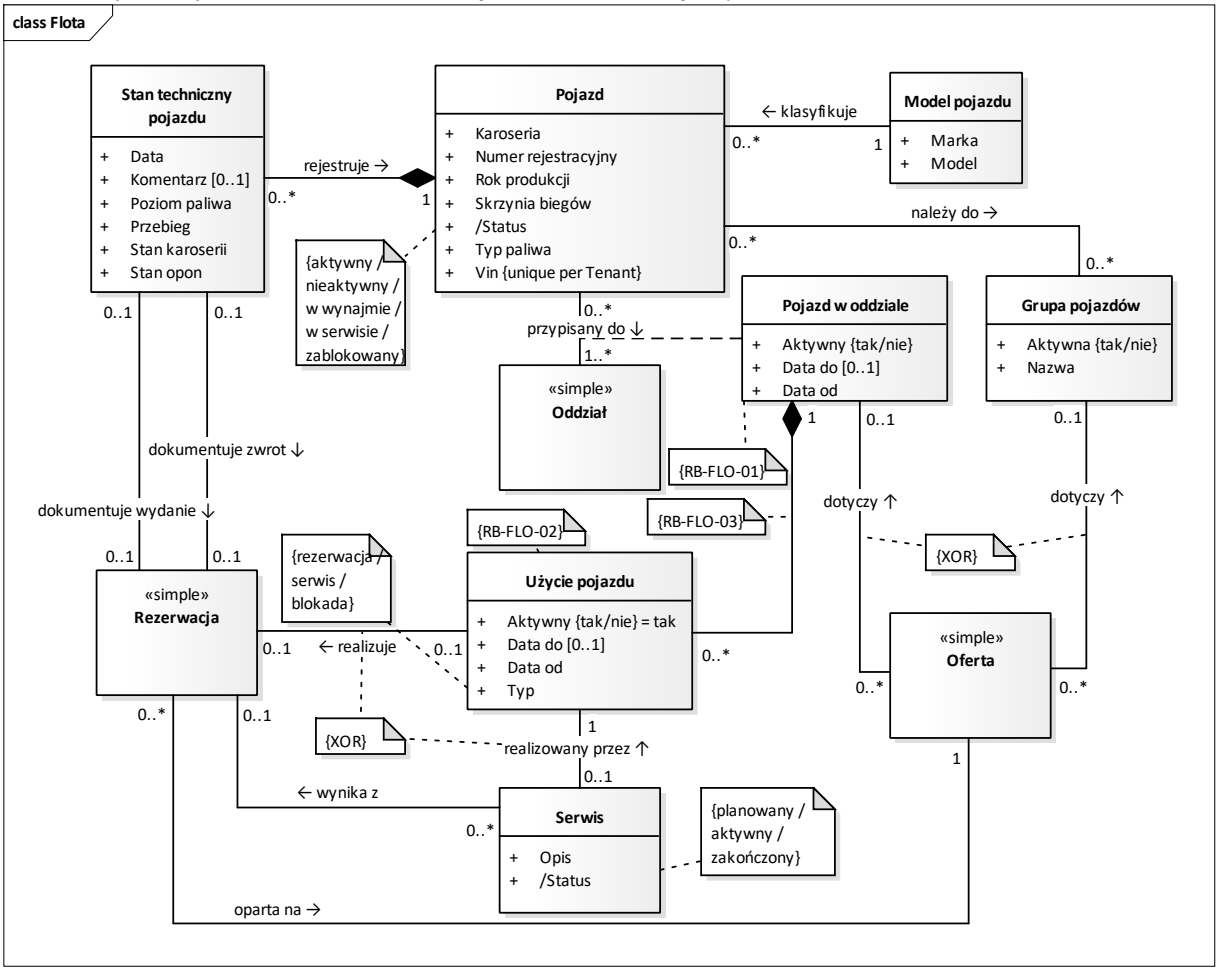
Obecność wypożyczalni w systemie reguluje subskrypcja, przy czym analitycznie istotne jest rozróżnienie dwóch pojęć. Pakiet to plan oferowany przez platformę, z limitami zasobów (oddziałów, pojazdów, pracowników) i czasem trwania, natomiast *Subskrypcja* to jego aktywacja u konkretnej wypożyczalni, z własnym okresem i statusem. Z powiązań tych wynikają reguły porządkujące dziedzinę: wypożyczalnia ma w danym momencie jedną aktywną subskrypcję, a jej zasoby mieszczą się w limitach wykupionego pakietu. Łańcuch odnowień oddaje samoreferencyjne powiązanie „następna subskrypcja” (0..1-0..1), wiążące subskrypcję wygasającą z zastępującą ją kolejną.

W kontekście wypożyczalni działają jej użytkownicy. Wspólną tożsamość kont, czyli dane uwierzytelniania oraz znaczniki stanu konta, wydzielono do abstrakcyjnego nadtypu *Użytkownik*, który uszczegóławiają trzy specjalizacje. *Administrator* reprezentuje administratora platformy, *Klient* to klient wypożyczalni rezerwujący pojazd (odróżniany danymi firmy, gdy jest podmiotem instytucjonalnym), a *Pracownik* to personel wypożyczalni, którego zakres działania określa atrybut rola (właściciel, biurowy lub operacyjny). Hierarchię tę należy odróżnić od hierarchii aktorów (podrozdział 5.2): tamta porządkuje dziedziczenie uprawnień, natomiast model dziedziny scala wspólną tożsamość kont w jednym nadtypie, a trzy role personelu ujmuje w jednej klasie rozróżnianej atrybutem. Gość oraz aktorzy techniczni nie występują jako klasy, ponieważ nie mają trwałej tożsamości w dziedzinie.

Powiązania łączą obie warstwy perspektywy: klient i pracownik należą do wypożyczalni, w której kontekście działają (administrator platformy pozostaje bytem globalnym, bez takiego przypisania), a pracownik dodatkowo do oddziału macierzystego. Pozycja rozliczenia występuje tu jedynie jako kontekst *(*<<*simple*>>), łączący tę perspektywę z finansami (punkt 5.7.4), gdzie rozliczenia wystawiane są dla klienta i przez oddział.

5.7.2. Flota

Perspektywa floty obejmuje pojazdy wraz z ich przypisaniem do oddziałów, stanem technicznym, użyciem oraz serwisem. Jej strukturę ilustruje *Rysunek 51*.



Rysunek 51 Analityczny diagram klas – fragment: perspektywa floty. Źródło: opracowanie własne.

Centralnym bytem jest *Pojazd*, egzemplarz floty o numerze VIN unikalnym w obrębie wypożyczalni, odróżniony od *Modelu pojazdu*, który zbiera cechy wspólne dla wielu egzemplarzy

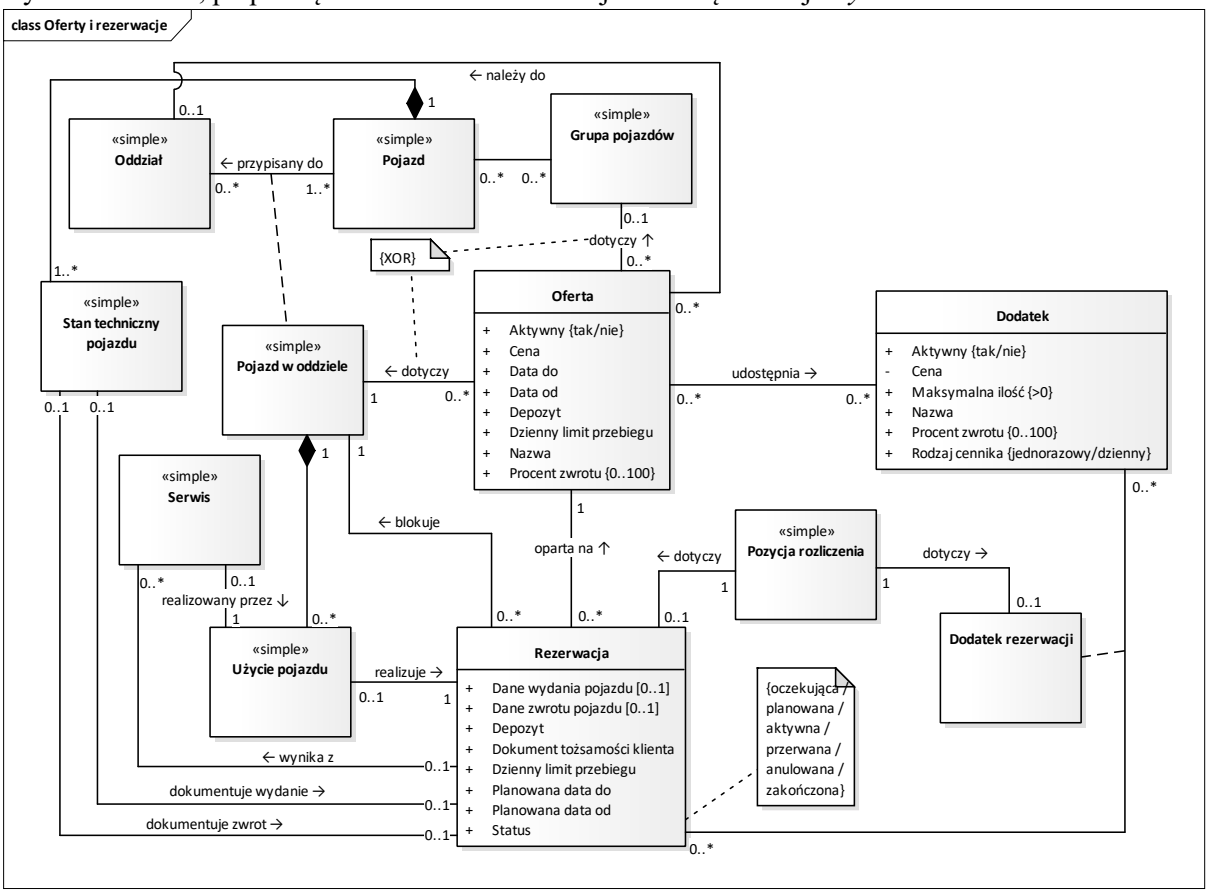
(markę i model). Pojazdy można łączyć w *Grupy pojazdów* (powiązanie wiele-do-wielu) wykorzystywane przy budowie ofert. Przypisanie pojazdu do oddziału ujęto jako klasę asocjacji *Pojazd w oddziale*, opisującą je w czasie: pojazd bywa relokowany między oddziałami, więc powiązanie to ma własny okres obowiązywania. *Oferta* dotyczy albo konkretnego pojazdu w oddziale, albo całej grupy pojazdów (ograniczenie *{xor}*).

Bieżące wykorzystanie pojazdu opisuje *Użycie pojazdu*, czyli okres jego zajętości o typie rezerwacja, serwis lub blokada. Rozróżnienie tych typów tłumaczy kluczową krotność perspektywy: powiązanie *Rezerwacja-Użycie* pojazdu jest opcjonalne po obu stronach (0..1-0..1), ponieważ jednemu użyciu odpowiada co najwyżej jedna rezerwacja, użyciami bywają jednak także niezwiązane z rezerwacją serwisy i blokady, a sama rezerwacja może nie mieć jeszcze przypisanego użycia.

Pojazd rejestruje swoje stany techniczne, czyli pomiary obejmujące między innymi przebieg i poziom paliwa, sporządzane w toku obsługi (kompozycja). Z rezerwacją łączą stan techniczny dwa odrębne powiązania, rozróżnione rolami „wydanie pojazdu" i „zwrot pojazdu", co pozwala zarejestrować stan pojazdu w obu tych momentach. *Serwis*, opisany statusem i zakresem prac, wiąże się z użyciem pojazdu oraz z rezerwacją. Klasy *Oferta*, *Rezerwacja* i *Oddział* występują tu jedynie jako konteksty *(<<simple>>)*, z pełnym opisem w punktach 5.7.1 oraz 5.7.3.

5.7.3. Oferty i rezerwacje

Perspektywa ofert i rezerwacji ujmuje rdzeń procesu najmu: od propozycji wynajmu, przez wybór dodatków, po powiązanie z rozliczeniem. Jej strukturę ilustruje *Rysunek 52*.



Rysunek 52 Analityczny diagram klas – fragment: perspektywa ofert i rezerwacji. Źródło: opracowanie własne.

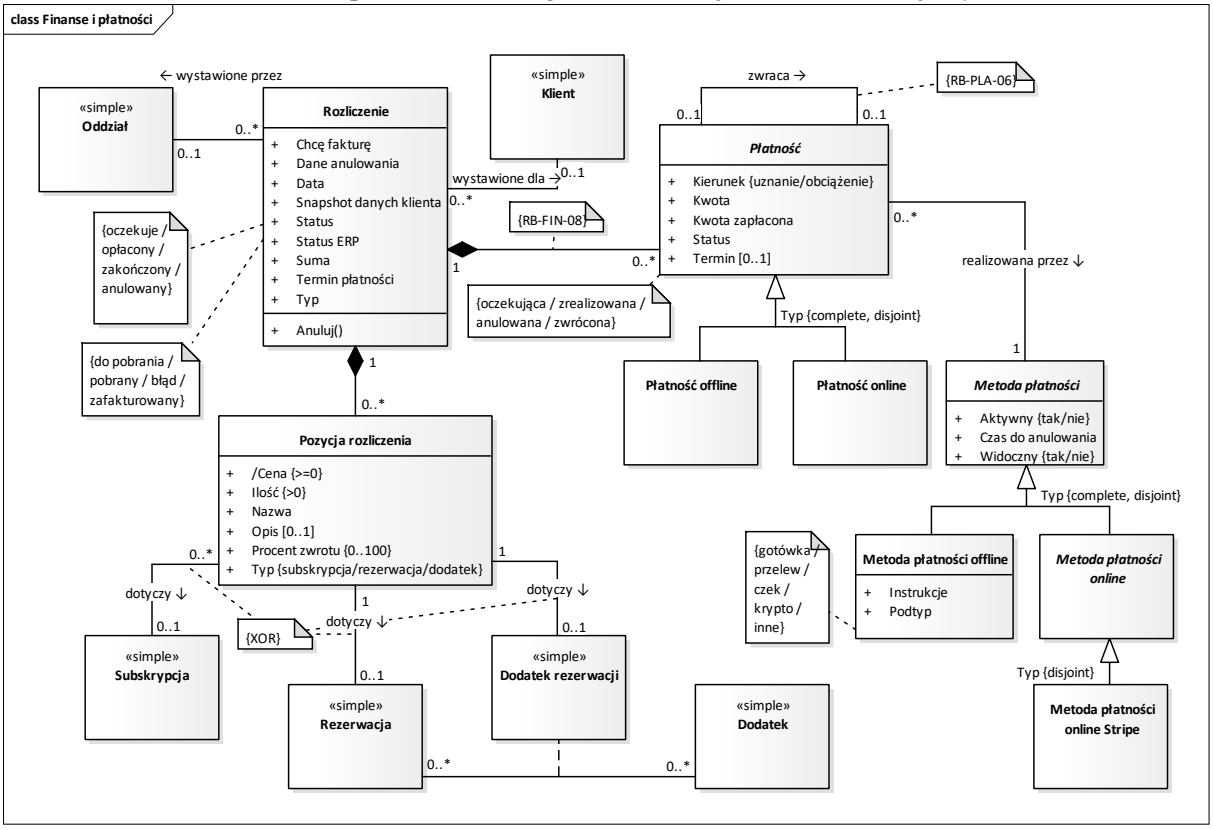
Oferta to propozycja najmu, określona warunkami wynajmu (okres ważności, dzienny limit przebiegu, depozyt) oraz ceną. Na jej podstawie powstaje *Rezerwacja*, która dotyczy dokładnie jednej

oferty i niesie własny status oraz dane konkretnego wynajmu: planowany okres, dokument tożsamości klienta oraz informacje o wydaniu i zwrocie pojazdu.

Kluczowym rozróżnieniem tej perspektywy jest podział dodatków na dostępne i wybrane. Oferta udostępnia katalog dostępnych dodatków (powiązanie wiele-do-wielu z *Dodatkiem*), z których każdy ma własny cennik i maksymalną ilość. Faktyczny wybór dodatku w danej rezerwacji modeluje natomiast klasa asocjacji *Dodatek rezerwacji* na powiązaniu *Rezerwacja-Dodatek*: to byt o własnym sensie (dodatek wybrany w konkretnej rezerwacji), odróżniony od dodatku jedynie oferowanego. Zarówno rezerwacja, jak i wybrany dodatek wiążą się z *Pozycją rozliczenia*, co łączy tę perspektywę z perspektywą finansową (punkt 5.7.4).

5.7.4. Finanse i płatności

Perspektywa finansów i płatności przedstawia, jak powstają rozliczenia i jak są opłacane, wraz z dziedziczeniem metod oraz płatności i obsługą zwrotów. Jej strukturę ilustruje *Rysunek 53*.



Rysunek 53 Analityczny diagram klas – fragment: perspektywa finansów i płatności. Źródło: opracowanie własne.

Dokumentem finansowym jest *Rozliczenie*, wystawiane dla klienta i przez oddział, opisane typem (sprzedaż lub zakup), sumą i statusem; osobny status ERP odzwierciedla synchronizację z zewnętrznym systemem, który odpowiada za fakturowanie. Rozliczenie składa się z *Pozycji rozliczenia* (kompozycja). Pozycja jest polimorficzna: rozlicza dokładnie jeden byt, czyli *Subskrypcję*, *Rezerwację* albo *Dodatek rezerwacji* (ograniczenie *{xor}*), i łączy perspektywę finansową z platformą (punkt 5.7.1) oraz z ofertami i rezerwacjami (punkt 5.7.3). Klient, oddział i rozliczane byty występują tu jako konteksty *(<<simple>>)*.

Opłaty modeluje *Płatność*, będąca częścią rozliczenia (kompozycja) i wykonywana określoną *Metodą płatności*. Obie rodziny pojęć porządkuje dziedziczenie: zarówno *Płatność*, jak i *Metoda płatności* dzielą się na offline i online. Pozwala to wspólnie traktować wszystkie płatności, zachowując zarazem specyfikę każdego kanału.

Zwroty oddaje samoreferencyjne powiązanie *Płatności* „zwraca” (0..1–0..1), w którym płatność zwrotna wskazuje płatność pierwotną.

5.8. Podsumowanie

Przeprowadzona analiza dostarczyła spójnego obrazu warstwy biznesowej systemu Wellhicle. Ustalono katalog aktorów porządkujący wszystkich uczestników systemu, opisano wymagania funkcjonalne w podziale na trzynaście modułów wraz z regułami biznesowymi i walidacjami, a także zebrano wymagania niefunkcjonalne ujmujące cechy jakościowe rozwiązania. Wymagania te zilustrowano przypadkami użycia oraz wybranymi scenariuszami z diagramami aktywności, a pojęcia dziedzinowe powracające w całym rozdziale uporządkowano w analitycznym modelu dziedziny. Powstał w ten sposób materiał, który nie opisuje jeszcze sposobu zbudowania systemu, lecz precyzyjnie określa, czego się od niego oczekuje.

Z analizy wyłania się kilka rozstrzygnięć nadających kształt całemu systemowi. Najważniejsze z nich to obsługa wielu niezależnych wypożyczalni w ramach jednej platformy oraz wyraźne oddzielenie klienta platformy od klienta wypożyczalni. Wynikają z nich dalsze ograniczenia biznesowe, w tym powiązanie dostępnych zasobów wypożyczalni z wykupionym pakietem subskrypcyjnym oraz zasada izolacji danych pomiędzy wypożyczalniami.

Reguły biznesowe sformułowane na etapie wymagań stanowią jednocześnie ograniczenia, które projekt systemu musi uwzględnić. Obejmują one między innymi wykluczanie nakładających się okresów rezerwacji tego samego pojazdu - tak aby nie mógł być on jednocześnie zarezerwowany, serwisowany lub zablokowany; wymóg utrzymywania przez każdego klienta platformy dokładnie jednej aktywnej subskrypcji; egzekwowanie limitów pakietu przy tworzeniu nowych pojazdów we flocie oraz odwracalność dezaktywacji kont, oddziałów i wypożyczalni z natychmiastowym unieważnianiem powiązanych sesji. Wymagania niefunkcjonalne (podrozdział *5.4*) wprowadzają dodatkowe ograniczenia przekrojowe - dotyczące bezpieczeństwa, wydajności, niezawodności i lokalizacji danych – które obowiązują wszystkie moduły systemu.

Tak zarysowane wymagania wyznaczają zadania dla kolejnego etapu pracy. Na poziomie projektowania dziedzina zostaje odwzorowana w projektowy model klas o strukturze bliskiej implementacji, opisane zostają architektura systemu oraz projekt interfejsu użytkownika, a reguły biznesowe znajdują rozstrzygnięcie co do cykli życia kluczowych bytów oraz miejsca ich egzekwowania - w warstwie domeny, aplikacji lub bazy danych. Zagadnienia wykraczające poza projekt, w tym sposób wdrożenia i utrzymania systemu, omówiono w dalszej, implementacyjnej części pracy.

Analiza pozostaje na poziomie biznesowym. Nie rozstrzyga szczegółów interfejsu użytkownika ani decyzji implementacyjnych, a wskazane reguły i wymagania opisują oczekiwane zachowanie systemu, nie zaś techniczny sposób jego osiągnięcia. Dzięki temu stanowi stabilny punkt odniesienia, do którego sięgają wszystkie późniejsze rozdziały dokumentu.

6. Projektowanie

Rozdział przedstawia etap projektowania systemu Wellhicle, stanowiący przejście od wyników analizy do opisu realizacji technicznej. Na podstawie zidentyfikowanych aktorów, zakresu funkcjonalnego, wymagań niefunkcjonalnych, przypadków użycia oraz modelu analitycznego określono rozwiązania projektowe, które porządkują strukturę systemu i sposób współpracy jego głównych elementów.

W tej części opisano najważniejsze decyzje dotyczące organizacji platformy SaaS, w tym wielotenantowość, obsługę wielu języków i walut, podział aplikacji frontendowych, integrację z systemem ERP oraz wykorzystanie aplikacji mobilnej. Następnie przedstawiono architekturę systemu, obejmującą komponenty odpowiedzialne za komunikację, przetwarzanie danych, uwierzytelnianie, obsługę plików, wiadomości, dokumentów oraz poczty elektronicznej.

6.1. Założenia projektowe

Ten podrozdział przedstawia założenia projektowe wypracowane w trakcie spotkań zespołu projektowego w początkowej fazie projektu. Założenia te określiły najważniejsze ograniczenia i kierunki rozwoju systemu Wellhicle, a następnie wpłynęły na decyzje architektoniczne oraz implementacyjne opisane w dalszej części pracy. Dotyczą one zarówno sposobu obsługi wielu wypożyczalni w ramach jednej platformy, jak i wymagań wynikających z planowanego wykorzystania systemu w różnych krajach, językach oraz modelach organizacyjnych.

6.1.1. Wielotenantowość

System obsługuje wielu najemców w ramach jednej współdzielonej instancji bazy danych. Wybrane tabele bazy zawierają kolumnę identyfikującą najemcę, której wartość przypisuje każdy rekord do konkretnej wypożyczalni. Serwer aplikacyjny dołącza warunek filtrowania po tym identyfikatorze do każdego zapytania dotyczącego danych najemcy, co uniemożliwia dostęp jednej wypożyczalni do danych innej [95].

Podejście z jedną instancją upraszcza administrację. Wymaga utrzymania jednego schematu, jednej konfiguracji kopii zapasowych i jednego procesu migracji. W porównaniu z modelem, w którym każdy najemca posiada osobną bazę danych, eliminuje to konieczność zarządzania wieloma instancjami i synchronizowania ich wersji schematu.

Rozwiązanie to niesie jednak istotne ograniczenia wydajnościowe. Wszyscy najemcy współdzielą te same zasoby bazy danych, przez co intensywne zapytania jednego klienta mogą podnosić czas odpowiedzi dla pozostałych. Przywrócenie danych konkretnego najemcy z kopii zapasowej wymaga odtworzenia całej bazy w środowisku tymczasowym i wyodrębnienia z niej wybranych rekordów, co jest procedurą czasochłonną i wymaga dodatkowej infrastruktury.

6.1.2. Wielojęzyczność

System Wellhicle jest architektonicznie przystosowany do obsługi dowolnych języków, jednak wdrożenie każdego nowego pakietu językowego wymaga analizy oraz ewentualnego dostosowania interfejsu użytkownika do lokalnych konwencji zapisu. W obecnej fazie rozwoju system oferuje zróżnicowany zakres wsparcia językowego w zależności od typu aplikacji:

- Aplikacje klienckie (do korzystania z usług) wspierają 35 języków.

- Aplikacje administracyjne do zarządzania wypożyczalniami są dostępne wyłącznie w języku angielskim.
- Aplikacje prezentujące ofertę systemu oraz służące do zarządzania flotą pojazdów obsługują język angielski i polski.

Taki podział wynika z faktu, że aplikacje klienckie mają bezpośredni kontakt z klientami wypożyczalni, dlatego kluczowe jest wsparcie jak najszerszej gamy języków.

6.1.3. Wielowalutowość

Platforma Wellhicle jest projektowana z myślą o globalnej gotowości do obsługi wszystkich krajów i walut. Wprowadzenie nowej waluty lub lokalizacji wymaga jednak analizy formatowania oraz odpowiedniego przygotowania komponentów interfejsu użytkownika, aby uniknąć błędów w wyświetlaniu danych.

W aktualnej konfiguracji aplikacja zapewnia pełne wsparcie dla 13 głównych walut, przy czym elastyczna struktura umożliwia przyszłe rozszerzanie tej listy o dodatkowe waluty.

6.1.4. Podział aplikacji *frontendowych* na 4 oddzielne serwery WWW

Decyzja o obsłudze czterech aplikacji *frontendowych* przez cztery oddzielne serwery WWW wynika z przyjętych zasad izolacji i niezależności wdrożeń.

Każda aplikacja działa we własnym, odseparowanym środowisku uruchomieniowym. Taki podział gwarantuje, że awaria jednego serwera nie wpływa na dostępność pozostałych. Dotyczy to w szczególności sytuacji, gdy intensywne operacje w jednym obszarze systemu mogłyby obniżyć stabilność części dostępnej dla klientów wypożyczalni.

Separacja umożliwia ponadto prowadzenie niezależnych cykli wdrożeniowych. Aktualizacja jednej aplikacji *frontendowej* nie wymaga ponownego uruchomienia pozostałych serwerów ani ingerencji w ich działanie. Skraca to czas wdrożenia i ogranicza ryzyko wprowadzenia regresji w niezmienianych częściach systemu.

Podział odpowiada również strukturze funkcjonalnej platformy. System obsługuje cztery odrębne obszary:

- aplikację do prezentacji oferty systemu,
- aplikację do zarządzania wypożyczalnią,
- aplikację do administrowania wypożyczalniami,
- aplikację do korzystania z usług wypożyczalni.

Każdy z tych obszarów ma inny zestaw użytkowników, uprawnień i wymagań dotyczących dostępności. Przypisanie każdego obszaru do osobnego serwera pozwala zarządzać nimi niezależnie, zarówno na etapie wdrożenia, jak i utrzymania.

Podział warstwy *frontendowej* został odzwierciedlony również na poziomie organizacji kodu źródłowego. Każda z czterech aplikacji przeglądarkowych jest rozwijana jako osobny projekt frontendowy w serwisie GitHub. W takim podejściu poszczególne aplikacje w warstwie prezentacji mogą być rozwijane, testowane i wdrażane niezależnie, zgodnie z własnym zakresem odpowiedzialności oraz grupą użytkowników.

Aby ograniczyć powielanie kodu, system wykorzystuje wspólną bibliotekę używaną przez wszystkie projekty frontendowe. Zawiera ona elementy współdzielone, takie jak komponenty

interfejsu, typy danych, mechanizmy komunikacji z API oraz funkcje pomocnicze używane w więcej niż jednej aplikacji. Rozwiązanie to pozwala zachować spójność wizualną i techniczną systemu, a jednocześnie nie wymusza łączenia wszystkich aplikacji w jeden duży projekt.

6.1.5. Integracja z systemem ERP

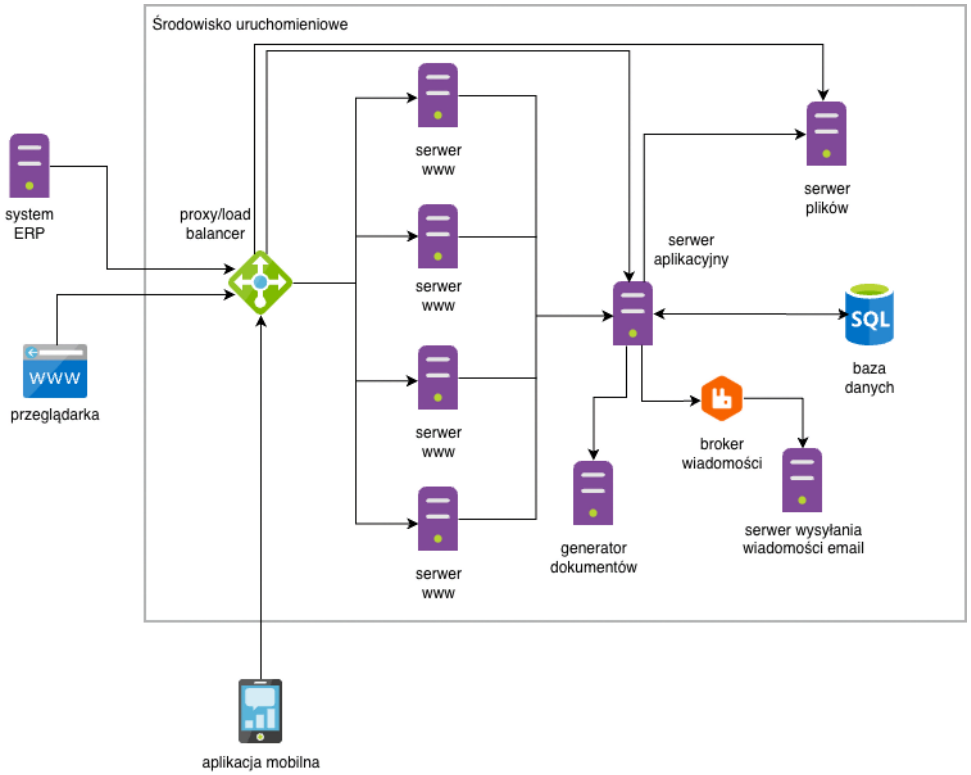
System Wellhicle udostępnia interfejs, za pośrednictwem którego system ERP może pobierać rozliczenia oraz płatności związane z zakupionymi subskrypcjami i zrealizowanymi wypożyczeniami. Po wygenerowaniu faktury w systemie ERP, dokument ten może zostać przesłany do Wellhicle celem zapisania na serwerze plików oraz udostępnienia klientowi wypożyczalni.

6.1.6. Wieloplatformowa aplikacja mobilna

Aplikacja mobilna, stanowiąca uzupełnienie systemu do zarządzania wypożyczalnią pojazdów Wellhicle, jest kompatybilna z dwoma dominującymi na rynku systemami operacyjnymi urządzeń mobilnych: iOS oraz Android. Dzięki temu nie narzuca konkretnego modelu urządzenia mobilnego pracownikom operacyjnym wypożyczalni, zapewniając elastyczność i uniwersalność użytkowania.

6.2. Architektura systemu

System Wellhicle składa się z kilku wyspecjalizowanych komponentów, których współpraca umożliwia obsługę zarówno klientów przeglądarkowych, aplikacji mobilnej jak i systemu ERP. *Rysunek 54* przedstawia pełny schemat architektury. W niniejszym podrozdziale opisane są poszczególne komponenty systemu oraz ich odpowiedzialności.



Rysunek 54 Diagram architektury systemu. Źródło: opracowanie własne.

6.2.1. Serwer *proxy*

Serwer *proxy* pełni rolę jedynego punktu wejściowego dla całego ruchu sieciowego kierowanego do systemu. Wszystkie żądania, zarówno z przeglądarek internetowych, jak i z aplikacji mobilnej, trafiają najpierw do tej warstwy, zanim zostaną przekazane do właściwych komponentów.

Podstawowym zadaniem serwera *proxy* jest rozdzielanie przychodzących żądań na podstawie zdefiniowanych reguł. W systemie Wellhicle zastosowano dwa odrębne schematy przekierowania:

- żądania z przeglądarki kierowane są do jednego z czterech serwerów WWW, w zależności od domeny.
- żądania z aplikacji mobilnej oraz z systemu ERP przekazywane są bezpośrednio do serwera aplikacyjnego, z pominięciem warstwy *frontendowej*.

Taki podział pozwala na precyzyjne sterowanie ruchem bez potrzeby modyfikacji logiki po stronie klientów.

Serwer *proxy* realizuje ponadto terminację SSL (ang. *SSL termination*). Szyfrowanie ruchu odbywa się wyłącznie na odcinku między klientem a serwerem *proxy*. Komunikacja wewnętrzna prowadzona jest bez szyfrowania, co upraszcza konfigurację komponentów wewnętrznych i eliminuje narzut związany z zarządzaniem certyfikatami w sieci wewnętrznej.

Zastosowanie modelu *„same-origin”* oznacza, że aplikacje *frontendowe* i serwer aplikacyjny dostępne są pod tą samą domeną, z rozróżnieniem na poziomie ścieżki. Polityka *„same-origin”* to mechanizm bezpieczeństwa przeglądarki ograniczający możliwość interakcji dokumentów i skryptów z zasobami pochodzącymi z innego źródła, gdzie *origin* wyznaczany jest przez kombinację protokołu, hosta i portu [96]. Dzięki temu przeglądarki traktują komunikację między aplikacją *frontendową* a serwerem aplikacyjnym jako ruch wewnątrz tego samego źródła, co eliminuje potrzebę konfigurowania mechanizmu CORS (ang. *Cross-Origin Resource Sharing*).

Podejście to ogranicza również powierzchnię podatną na ataki typu CSRF (*ang. Cross-Site Request Forgery*). Atak ten polega na tym, że przeglądarka ofiary zostaje nakłoniona do wysłania sfałszowanego żądania HTTP do aplikacji, której ufa, wraz z ciasteczkami sesji i innymi danymi uwierzytelniającymi [97]. Gdy *frontend* i *backend* działają w ramach tego samego źródła (*ang. origin)*, przeglądarka odmawia wykonania takich żądań ze stron trzecich, co stanowi pierwszą warstwę ochrony przed tym rodzajem ataku.

Serwer *proxy* obsługuje również subdomeny klientów systemu. Żądania kierowane na dynamiczne subdomeny są przepisywane na odpowiadające im ścieżki wewnętrzne, które następnie odczytuje aplikacja do korzystania z usług wypożyczalni. Konfiguracja opiera się na dynamicznym mapowaniu ścieżki zapytania do najemcy i nie wymaga ręcznego dodawania wpisów dla każdego nowego klienta.

Szczegóły implementacji serwera *proxy* zostały opisane w podrozdziale *7.4*.

6.2.2. Serwer WWW

Warstwa *frontendowa* systemu składa się z czterech serwerów WWW, z których każdy obsługuje jedną, wydzieloną aplikację przeglądarkową. Zadaniem każdego serwera jest udostępnianie plików statycznych tworzących interfejs użytkownika.

Serwery WWW nie realizują żadnej logiki biznesowej. Ich odpowiedzialność ogranicza się do dostarczenia kodu aplikacji do przeglądarki użytkownika. Całość komunikacji z serwerem aplikacyjnym odbywa się następnie bezpośrednio z poziomu przeglądarki, za pośrednictwem serwera *proxy*.

Ruch kierowany do serwerów WWW pochodzi wyłącznie od klientów przeglądarkowych. Aplikacja mobilna nie korzysta z tej warstwy, ponieważ komunikuje się bezpośrednio z serwerem aplikacyjnym.

6.2.3. Serwer aplikacyjny

Serwer aplikacyjny stanowi centralny komponent systemu odpowiedzialny za realizację logiki biznesowej. Przetwarza żądania przychodzące zarówno od klientów przeglądarkowych, jak i od aplikacji mobilnej, a następnie zwraca odpowiedzi w formacie uzgodnionym przez kontrakt API.

Serwer aplikacyjny komunikuje się z pozostałymi komponentami systemu w następujący sposób:

- odczytuje i zapisuje dane w bazie danych,
- przesyła pliki biznesowe na dedykowany serwer plików,
- publikuje komunikaty do brokera wiadomości, zlecając wykonanie zadań asynchronicznych,
- zleca zadania generatorowi dokumentów.

Zadania, które nie wymagają natychmiastowej odpowiedzi dla klienta, są delegowane do procesów roboczych za pośrednictwem brokera wiadomości. Należy do nich wysyłka wiadomości e-mail. Takie podejście pozwala skrócić czas odpowiedzi serwera i odciążyć go od operacji czasochłonnych.

Szczegóły implementacji serwera aplikacyjnego zostały opisane w podrozdziale *7.1*.

6.2.4. Baza danych

Zastosowano relacyjny model danych, ponieważ dane systemu mają wyraźnie zdefiniowaną strukturę i silne powiązania między encjami. Model relacyjny umożliwia wyrażenie tych powiązań za pomocą kluczy obcych i więzów integralności, co ogranicza ryzyko niespójności danych na poziomie schematu.

Baza danych zapewnia transakcyjność zgodną z modelem ACID, definiowanym jako zbiór czterech właściwości transakcji bazodanowej: atomowości (ang. *atomicity*), spójności (ang. *consistency*), izolacji (ang. *isolation*) oraz trwałości (ang. *durability*) [98]. Właściwości te są wymagane przy operacjach takich jak rezerwacje pojazdów i zmiany statusów floty, gdzie częściowe wykonanie operacji mogłoby prowadzić do błędów biznesowych.

Migracje bazy danych stosowane są automatycznie podczas uruchamiania nowej wersji serwera aplikacyjnego. Schemat bazy danych jest w ten sposób zawsze zsynchronizowany z wersją wdrożonego kodu.

Dostęp do bazy realizowany jest za pośrednictwem mapera obiektowo-relacyjnego (ORM). Jest to technika programistyczna polegająca na przekształcaniu danych między relacyjną bazą danych a obiektami stosowanymi w kodzie aplikacji [99]. Eliminuje to konieczność ręcznego tworzenia zapytań SQL w typowych przypadkach i oddziela logikę biznesową od szczegółów warstwy persystencji danych.

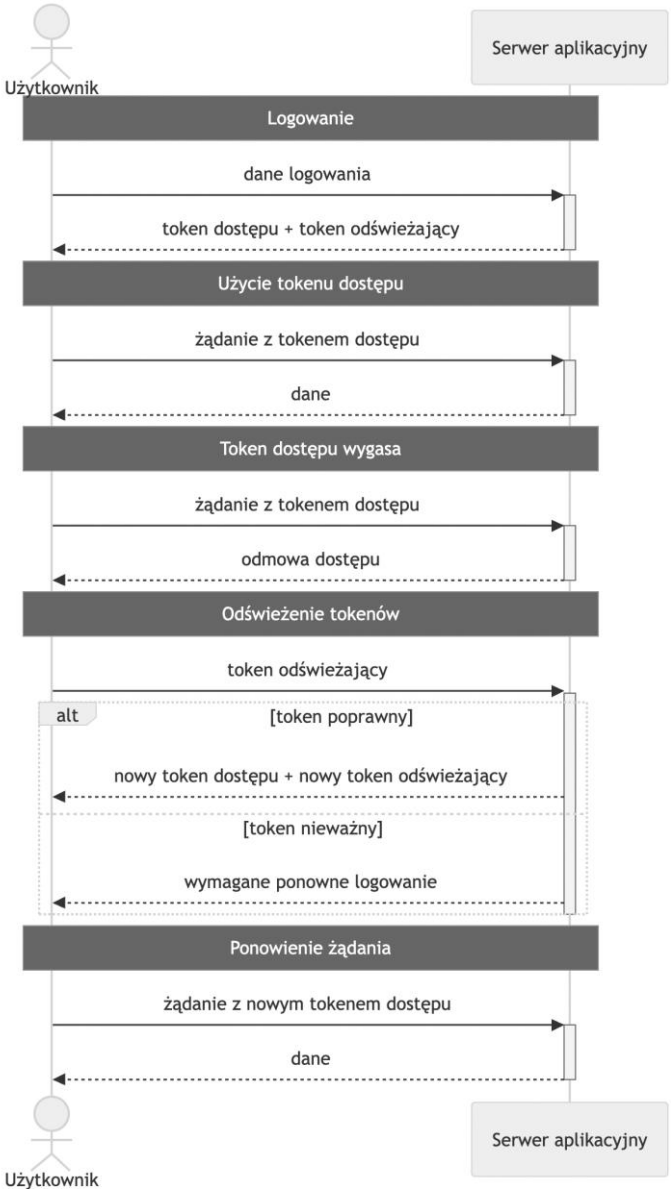
Połączenia z bazą danych zarządzane są przez mechanizm puli połączeń, który utrzymuje zestaw otwartych połączeń wielokrotnie wykorzystywanych przez kolejne żądania [100]. Ogranicza to obciążenia bazy danych związane z nawiązywaniem nowych połączeń i poprawia czas odpowiedzi systemu pod obciążeniem.

Szczegóły konfiguracji bazy danych zostały opisane w punkcie *7.1.4*.

6.2.5. Uwierzytelnianie i autoryzacja

Zastosowano bezstanowy mechanizm uwierzytelnienia oparty na tokenach JWT (ang. *JSON Web Token*), definiowanych jako zwarty, bezpieczny dla URL sposób reprezentowania twierdzeń przesyłanych między dwiema stronami [41]. Po poprawnym zalogowaniu serwer aplikacyjny wystawia dwa tokeny: token dostępowy (ang. *access token*) o krótkim czasie ważności oraz token odświeżający (ang. *refresh token*) o dłuższym czasie ważności. Klient dołącza token dostępowy do każdego żądania w nagłówku *HTTP Authorization*.

Gdy token dostępowy wygaśnie, można uzyskać nowy, przesyłając token odświeżający. Token odświeżający jest przechowywany po stronie serwera, co umożliwia jego unieważnienie, na przykład przy wylogowaniu lub zmianie hasła. Mechanizm odświeżenia tokenu dostępowego został przedstawiony na *Rysunek 55*.



Rysunek 55 Diagram ilustrujący mechanizm odświeżania tokenu dostępowego. Źródło: opracowanie własne.

Oprócz tokenu sesyjnego system wykorzystuje tokeny JWT do weryfikacji adresu e-mail oraz resetowania hasła. Tokeny te mają odrębny zakres i nie mogą być użyte do uwierzytelnienia w API.

Szczegóły uwierzytelniania i autoryzacji zostały opisane w podrozdziale *7.1*.

6.2.6. Serwer plików

Pliki biznesowe systemu, takie jak zdjęcia pojazdów, szablony dokumentów oraz wygenerowane umowy, przechowywane są na dedykowanym serwerze plików. Oddzielenie składowania plików od serwera aplikacyjnego wynika z różnicy w charakterze danych: pliki są obiektami binarnymi o zmiennym rozmiarze, których przechowywanie w relacyjnej bazie danych byłoby nieefektywne.

Serwer udostępnia dwa logicznie oddzielone obszary składowania: publiczny, dostępny bezpośrednio przez przeglądarkę, oraz prywatny, dostępny po wygenerowaniu tymczasowego klucza dostępu. Podział ten pozwala udostępniać zasoby takie jak zdjęcia pojazdów bez pośrednictwa serwera aplikacyjnego, przy jednoczesnej ochronie dokumentów wrażliwych, do których dostęp wymaga każdorazowej autoryzacji.

Szczegóły konfiguracji serwera plików zostały opisane w podrozdziale *7.4*.

6.2.7. Broker wiadomości

Broker wiadomości pośredniczy w komunikacji między serwerem aplikacyjnym a procesami roboczymi. Zadania, które nie wymagają natychmiastowej odpowiedzi dla klienta, są publikowane jako komunikaty do kolejki i przetwarzane asynchronicznie. Takie podejście skraca czas odpowiedzi serwera aplikacyjnego i oddziela inicjowanie zadania od jego wykonania.

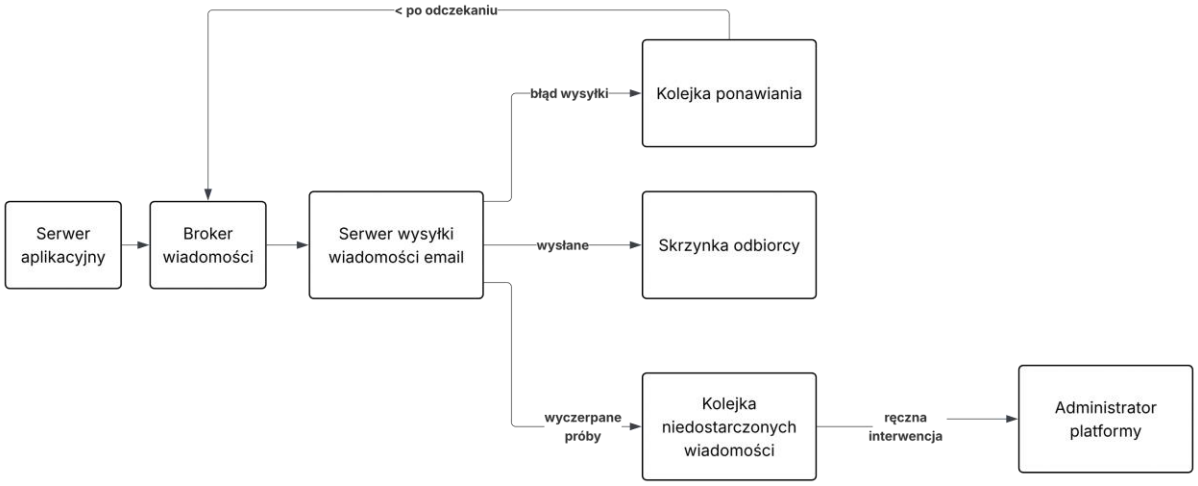
Komunikaty są trwałe, co gwarantuje ich zachowanie w przypadku restartu brokera. Zadania, których nie uda się przetworzyć po wyczerpaniu dopuszczalnej liczby prób, trafiają do kolejki niedostarczonych wiadomości (ang. *dead letter queue*), skąd mogą być analizowane lub ponownie przetworzone.

Szczegóły konfiguracji brokera wiadomości zostały opisane w podrozdziale *7.1*.

6.2.8. Serwer wysyłania wiadomości e-mail

Serwer wysyłania wiadomości e-mail działa jako niezależny proces roboczy, który odbiera zlecenia z brokera wiadomości i wysyła wiadomości do odbiorców. Wydzielenie tego procesu ze serwera aplikacyjnego pozwala na niezależne skalowanie i izoluje awarie związane z dostępnością serwera pocztowego od głównej części systemu.

Treść wiadomości opiera się na szablonach, które każda wypożyczalnia może dostosować do własnych potrzeb. W przypadku niedostępności serwera pocztowego nieudane próby są ponawiane, a po wyczerpaniu limitu wiadomość trafia do kolejki niedostarczonych wiadomości. *Rysunek 56* przedstawia przepływ wysyłki wiadomości e-mail w systemie Wellhicle.



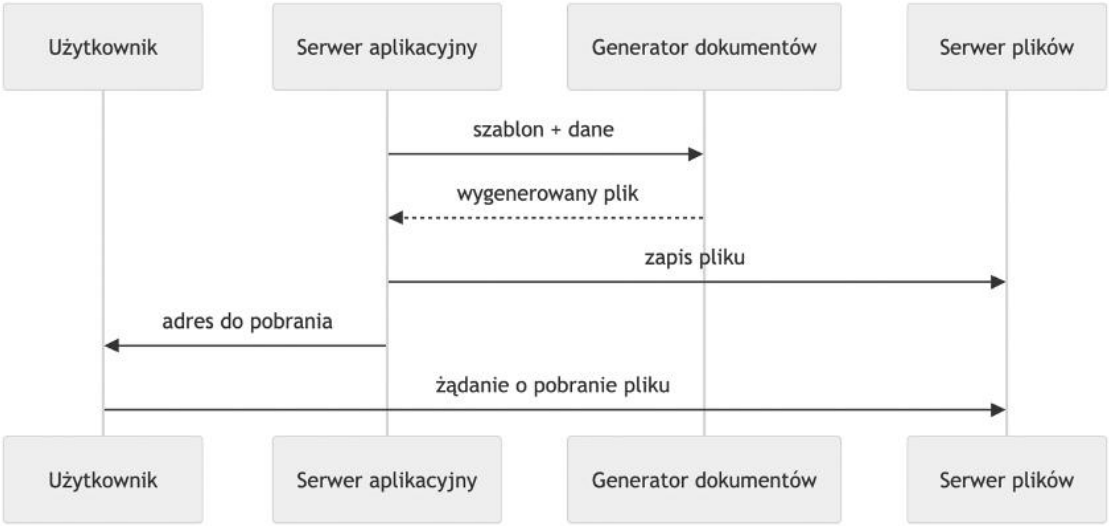
Rysunek 56 Diagram komponentów ilustrujący przepływ wysyłki wiadomości e-mail Źródło: opracowanie własne.

Szczegóły implementacji serwera wysyłania wiadomości e-mail zostały opisane w podrozdziale *7.1*.

6.2.9. Generator dokumentów

Generator dokumentów odpowiada za tworzenie plików PDF na podstawie szablonów. Serwer aplikacyjny przekazuje szablon wraz z danymi kontekstowymi do wyspecjalizowanego serwisu, który uzupełnia szablon i zwraca gotowy dokument. Wygenerowany plik jest zapisywany na serwerze plików i powiązany z odpowiednim bytem w bazie danych.

Szablony mogą być definiowane na poziomie platformy jako szablony globalne lub przez każdą wypożyczalnię jako szablony własne. Przed wygenerowaniem dokumentu system weryfikuje kompletność wymaganych danych, co zapobiega tworzeniu niekompletnych dokumentów. *Rysunek 57* przedstawia przepływ procesu generowania dokumentu w systemie Wellhicle.



Rysunek 57 Diagram ilustrujący przepływ procesu generowania dokumentu w systemie Wellhicle Źródło: opracowanie własne.

Szczegóły implementacji generatora dokumentów zostały opisane w podrozdziale *7.1*.

6.2.10. Aplikacja mobilna

Aplikacja mobilna odpowiada za obsługę funkcjonalności związanych z odbiorem oraz wydaniem pojazdu, a także ze zgłaszaniem serwisów. Komunikuje się bezpośrednio z serwerem aplikacyjnym, pomijając przy tym serwery WWW udostępniające statyczny kod aplikacji *frontendowych*.

Opis implementacji aplikacji mobilnej znajduje się w podrozdziale *7.3*.

6.3. Projektowy diagram klas

Podrozdział prezentuje projektowy model klas (*Project Class Diagram*, dalej PCD) warstwy domenowej systemu Wellhicle. Model w notacji UML opisuje strukturę warstwy domenowej na poziomie projektowym – klasy, ich atrybuty i operacje, relacje między nimi oraz ograniczenia integralności. Decyzje projektowe utrwalone w modelu uwzględniają wybór wzorców Domain-Driven Design oraz konsekwencje przyjętej platformy implementacyjnej (.NET wraz z Entity Framework Core) – uzasadnienie obu decyzji opisane jest w podrozdziale *7.1*.

W odróżnieniu od analitycznego modelu dziedziny prezentowanego w podrozdziale *5.7*, który operuje pojęciami biznesowymi po polsku i koncentruje się na zrozumieniu domeny, PCD opisuje strukturę warstwy domenowej w postaci gotowej do implementacji – z technicznymi typami, jawnymi relacjami, ograniczeniami integralności i polimorfizmem. Oba modele uzupełniają się: pierwszy odpowiada na pytanie jak rozumiemy dziedzinę, drugi – jak zaprojektowano jej strukturę.

Konwencja językowa

Diagramy oraz nazwy elementów modelu prezentowane są w języku angielskim – jest to konwencja przyjęta dla modelu projektowego. Typy atrybutów odpowiadają typom platformy .NET (`int`, `string`, `DateTime`, `bool`); nullowalność oznaczana jest znakiem `?` (np. `string?`).

Stosowane stereotypy

W modelu wykorzystano dwa stereotypy UML rozszerzające podstawową semantykę.

<<*simple*>> stosujemy w znaczeniu przyjętym w podrozdziale *5.7* – na diagramach modułowych oznacza on klasę spoza prezentowanego modułu.

<<*flags*>> oznacza typ wyliczeniowy z atrybutem *[Flags]* języka C# – wartości takiej enumeracji można łączyć za pomocą bitowego operatora OR. W modelu Wellhicle stereotyp ten odnosi się wyłącznie do *FuelType*, gdzie pojazd hybrydowy reprezentowany jest jako kombinacja paliw (np. *PB | EV* dla hybrydy benzynowo-elektrycznej).

Stosowane ograniczenia

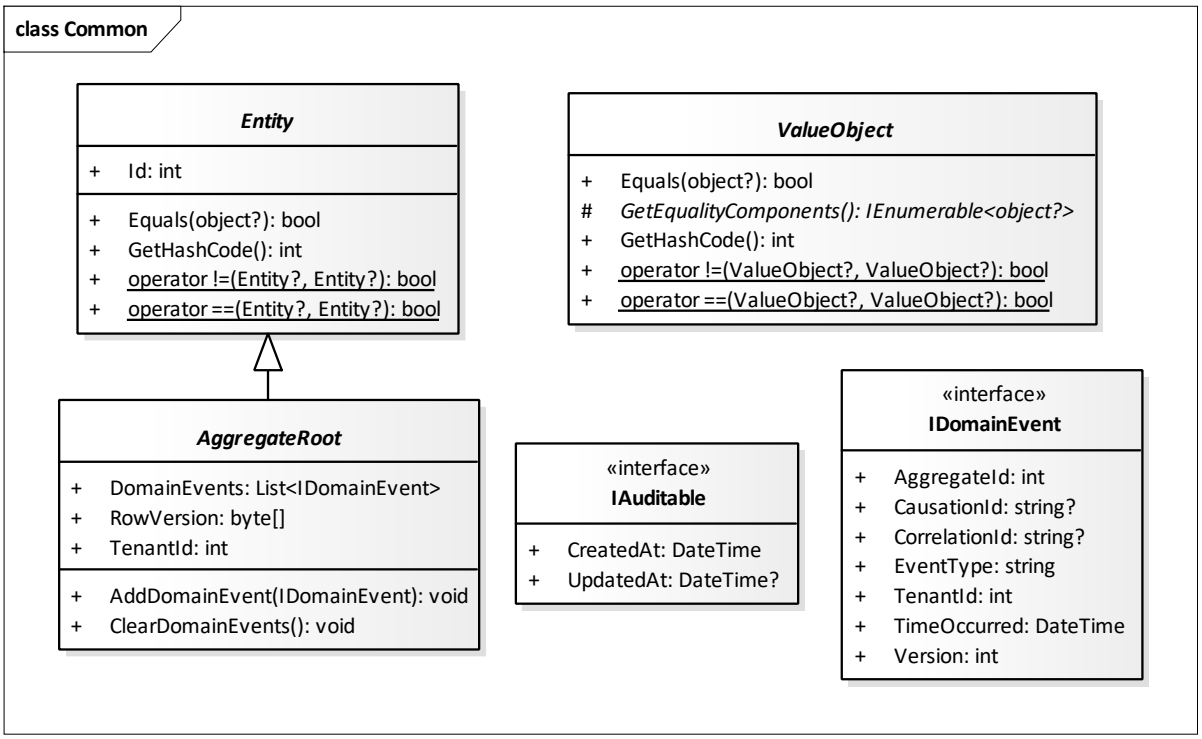
Na diagramach wykorzystano standardowe ograniczenia UML zapisywane w nawiasach klamrowych przy atrybutach oraz oznaczenie *XOR* przy rozłącznych alternatywach asocjacji. Poza notacją standardową wprowadzono jeden wariant własny: *{unique per Tenant}* – unikalność obowiązująca w obrębie pojedynczego tenanta, typowa dla wielonajemczego SaaS, gdzie ten sam zbiór wartości może wystąpić u innego klienta platformy.

Struktura podrozdziału

Podrozdział *6.3* zawiera łącznie 23 diagramy zorganizowane w cztery warstwy: *6.3.1* prezentuje wspólne podstawy domeny, *6.3.2* – katalog obiektów wartości, *6.3.3* – dwanaście diagramów modułowych pokrywających pełną warstwę domenową, *6.3.4* – dziewięć diagramów typów wyliczeniowych pogrupowanych tematycznie.

6.3.1. Wspólne podstawy domeny

Diagram modułu Common z *Rysunek 58* przedstawia warstwę abstrakcji wykorzystywaną przez wszystkie pozostałe diagramy modułowe podrozdziału *6.3* – trzy klasy bazowe (*Entity*, *AggregateRoot*, *ValueObject*) oraz dwa interfejsy techniczne (*IAuditable*, *IDomainEvent*). Pełni funkcję legendy strukturalnej: diagramy modułowe nie powtarzają klas bazowych ani interfejsów technicznych jako odrębnych boksów, a dziedziczenie sygnalizowane jest jedynie nazwą bazy w nagłówku klasy potomnej.



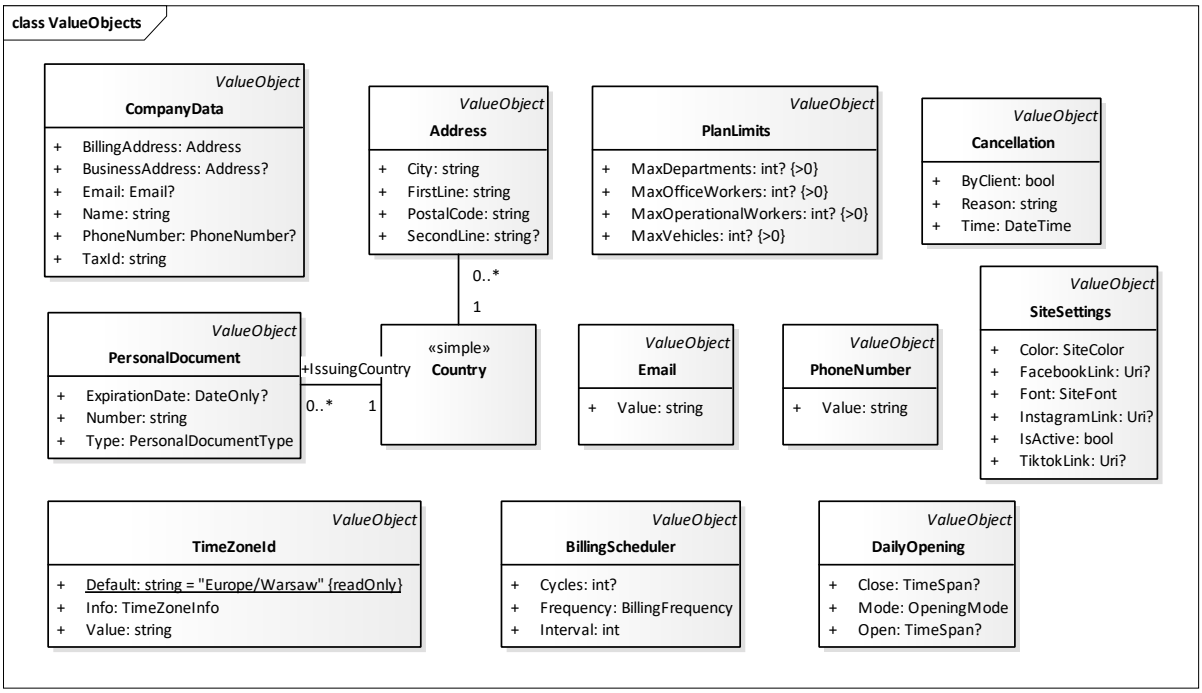
Rysunek 58 Wspólne podstawy domeny – klasy bazowe i interfejsy techniczne wykorzystywane przez wszystkie pozostałe diagramy PCD. Źródło: opracowanie własne.

Interfejs *IAuditable* jest implementowany selektywnie. Nie implementują go: słowniki referencyjne (*Currency*, *Country*, *Language*), klasy niemodyfikowalne po wdrożeniu (*Config*, *Translation*, *VehicleModel*, *MergeFieldDefinition*), klasa *SmtpAccount* oraz obiekty wartości (z natury niepodlegające audytowi). Decyzja o pominięciu interfejsu na diagramach modułowych podyktowana była ekonomią miejsca – implementacja przez ponad dwadzieścia klas powodowałaby powtarzanie tej samej informacji w niemal każdym punkcie.

Szczegółowy opis klas bazowych oraz mechanizmów ich zachowania (równość encji, optymistyczna kontrola współbieżności, rejestracja i publikacja zdarzeń domenowych, automatyczne wypełnianie pól audytowych) znajduje się w podrozdziale *7.1*.

6.3.2. Obiekty wartości

Rysunek 59 prezentuje katalog obiektów wartości (*ValueObject*) systemu Wellhicle. Wyodrębnienie tych klas na osobny diagram pozwala uniknąć powtarzania ich definicji na PCD modułowych – na diagramach modułowych obiekty wartości pojawiają się wyłącznie jako typy atrybutów (np. *User.Email : Email*, *Tenant.Company : CompanyData*).



Rysunek 59 Katalog obiektów wartości systemu Wellhicle. Źródło: opracowanie własne.

Diagram zawiera obiekty wartości wykorzystywane przekrojowo w modelu domeny: *Email*, *PhoneNumber*, *Address*, *TimeZoneId*, *CompanyData* (dane firmy klienta lub tenanta), *PersonalDocument* (dokument tożsamości klienta), *BillingScheduler* (harmonogram cykli rozliczeniowych subskrypcji), *PlanLimits* (limity funkcjonalne planu subskrypcyjnego), *Cancellation* (dane anulowania dokumentu lub rezerwacji), *DailyOpening* (godziny otwarcia oddziału w danym dniu tygodnia) oraz *SiteSettings* (ustawienia witryny tenanta).

Mechanizmy walidacji i porównywania obiektów wartości opisane są w podrozdziale *7.1.*

6.3.3. Diagramy modułowe

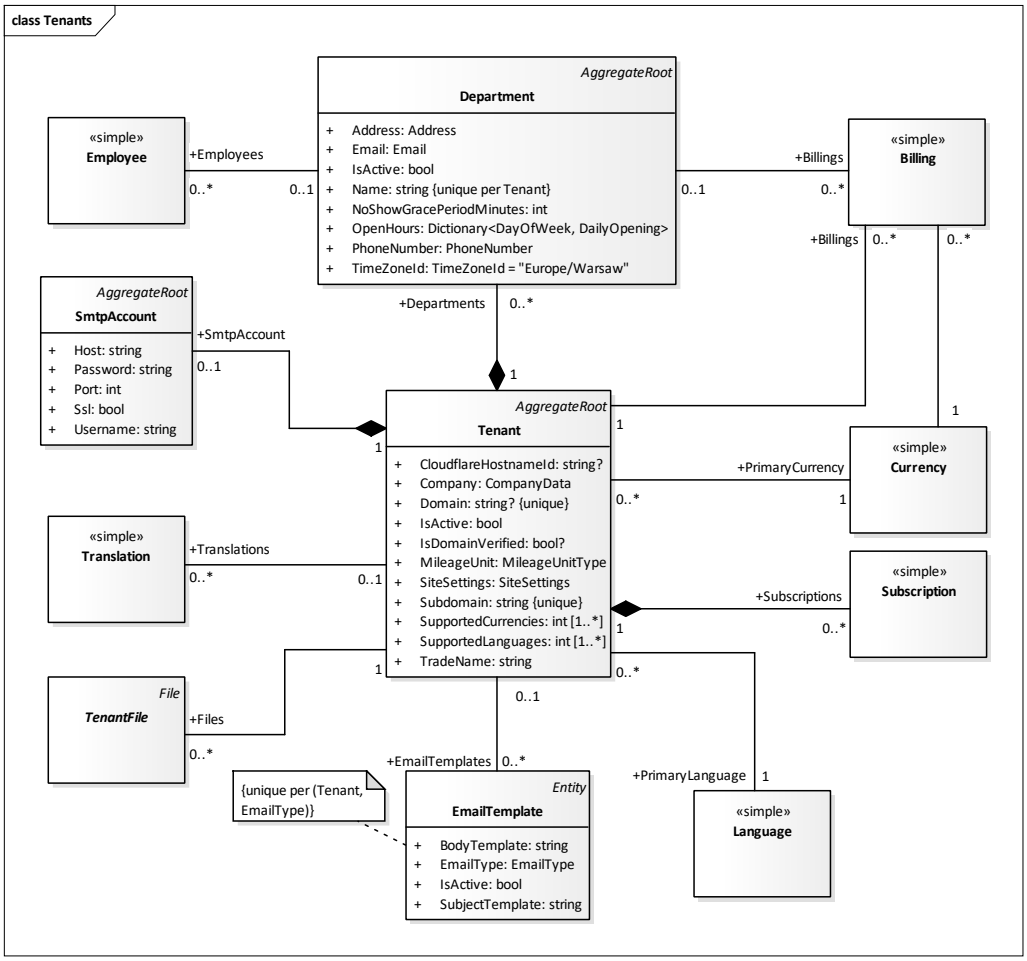
Punkt zawiera dwanaście diagramów modułowych pokrywających pełną warstwę domenową. Podział na osobne diagramy wynika z naturalnych granic modułów dziedziny oraz z ograniczeń czytelności – pełen zbiór encji domeny na jednym diagramie nie zmieściłby się na pojedynczej stronie i zaciemniłby relacje istotne wewnątrz każdego modułu.

Większość diagramów odpowiada pojedynczemu modułowi domeny (Tenants, Users, Files, ReferenceData, Configs, Translations, Vehicles, Subscriptions, Billings, Payments). Diagram Offers & Reservations *6.3.3.9* łączy dwa pokrewne moduły – Offers i Reservations – z uwagi na ścisłe powiązanie ofert z ich realizacją w postaci rezerwacji. Diagram Products *6.3.3.8* prezentuje przekrojowy widok wokół interfejsu *IProduct* i trzech klas implementujących (*Offer*, *Addition*, *Plan*) pochodzących z różnych modułów. Klasy z mniejszych modułów (*Emails*, *Departments*) zostały włączone do diagramu Tenants (odpowiednio *EmailTemplate*, *SmtpAccount* oraz *Department*), zgodnie z ich biznesowym kontekstem użycia.

6.3.3.1. Tenants

Moduł Tenants przedstawiony na *Rysunek 60* prezentuje agregat tenanta – najemcy platformy SaaS – wraz z powiązanymi encjami konfiguracyjnymi: oddziałami wypożyczalni (*Department*),

szablonami wiadomości e-mail (*EmailTemplate*), konfiguracją SMTP (*SmtpAccount*) oraz abstrakcyjną klasą plików tenantowych (*TenantFile*).



Rysunek 60 PCD modułu Tenants – agregat tenanta i powiązane encje konfiguracyjne. Źródło: opracowanie własne.

Na atrybucie *Tenant.Subdomain* zaznaczono ograniczenie *{unique}* – subdomena identyfikuje instancję wypożyczalni w obrębie platformy i jest podstawą routingu żądań HTTP. Atrybut *Tenant.Domain* o typie string? posiada warunkowe ograniczenie *{unique}* – obowiązuje wyłącznie dla rekordów z ustawioną wartością; rekordy bez własnej domeny (null) nie podlegają sprawdzeniu.

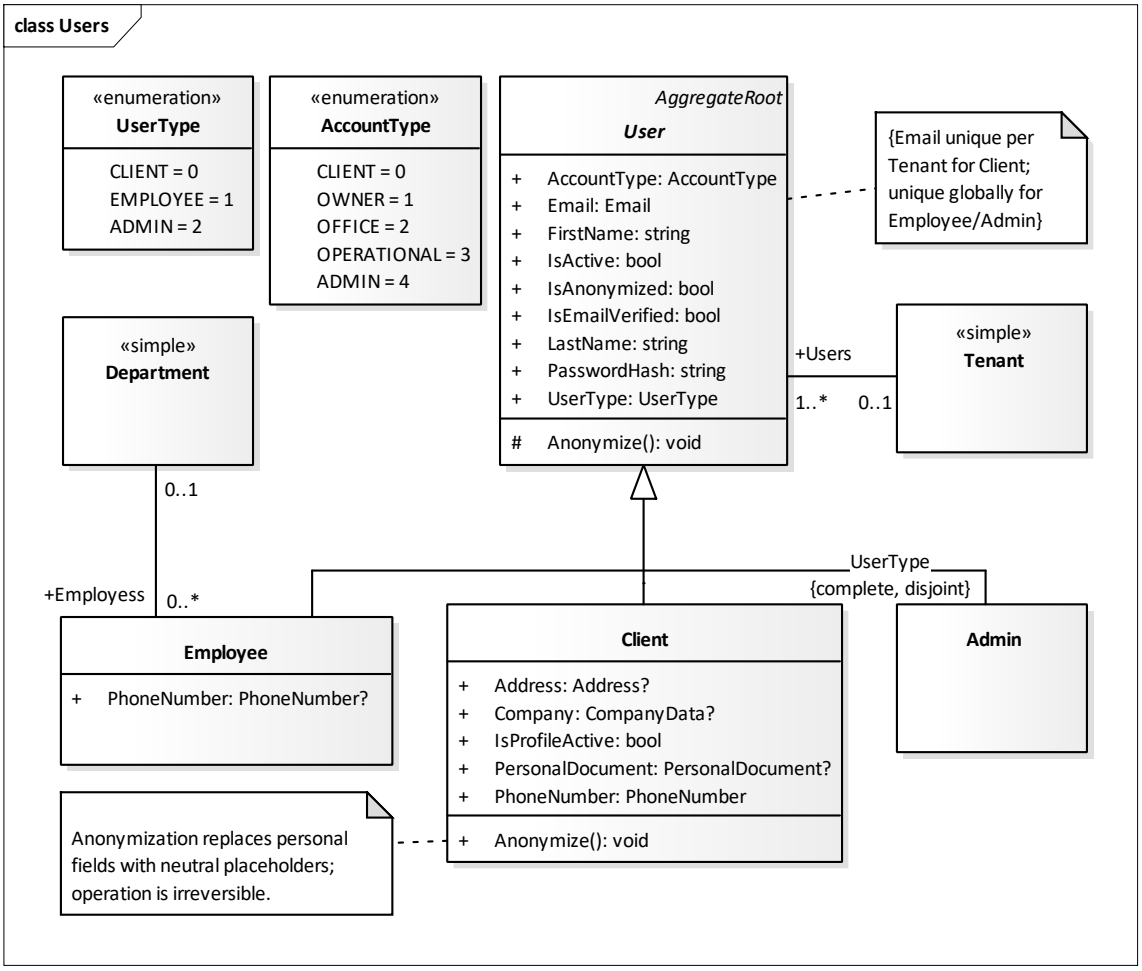
Szablony EmailTemplate dzielą się na globalne (wspólne dla całej platformy) i lokalne (powiązane z konkretnym tenantem) – szablony lokalne nadpisują szablony globalne tego samego typu. Ograniczenie widoczne na diagramie egzekwuje regułę unikalności w obrębie tenanta: nie może istnieć więcej niż jeden aktywny szablon danego typu (*EmailType*) dla pojedynczego klienta.

Tenant przechowuje listy obsługiwanych walut i języków (*SupportedCurrencies*, *SupportedLanguages* – atrybuty typu `int[]`) wraz z wyróżnionymi wartościami podstawowymi (*PrimaryCurrency*, *PrimaryLanguage* – klasyczne klucze obce odnoszące się do encji słownikowej). Listy obsługiwanych zbiorów nie są realizowane za pomocą typowej tabeli pomocniczej dla relacji wiele-do-wielu, lecz utrwalane jako pojedyncze kolumny tekstowe zawierające zserializowaną tablicę identyfikatorów. Takie rozwiązanie upraszcza strukturę bazy danych – eliminuje osobną tabelę powiązań, kosztem braku kontroli spójności referencyjnej na poziomie bazy danych. Pełna materializacja encji słownikowych po identyfikatorach z listy wymaga dodatkowego zapytania w warstwie aplikacji, w której również realizowana jest walidacja istnienia tych identyfikatorów w odpowiednich słownikach.

Wymagania funkcjonalne dotyczące konfiguracji tenanta oraz zasobów konfiguracyjnych zostały opisane w punktach 5.3.1, 5.3.11 i 5.3.12.

6.3.3.2. Users

Moduł Users przedstawiony na *Rysunek 61* prezentuje hierarchię kont użytkowników systemu. Abstrakcyjna klasa bazowa *User* definiuje pola wspólne dla wszystkich kont, a trzy klasy konkretne – *Client*, *Employee* i *Admin* – odpowiadają trzem kategoriom kont wyznaczonym w punkcie *5.3.3.* Klasa *Client* zawiera atrybuty specyficzne dla profilu klienta wypożyczalni, w tym dwie niezależne flagi aktywności: *IsActive* dla konta oraz *IsProfileActive* dla samego profilu – rozdzielenie to umożliwia dezaktywację profilu bez utraty historycznych powiązań konta z rezerwacjami i dokumentami rozliczeniowymi.



Rysunek 61 PCD modułu Users – hierarchia kont użytkowników z polimorfizmem przez dziedziczenie. Źródło: opracowanie własne.

Hierarchia dziedziczenia utrwalana jest strategią *Table-per-Type* – każda klasa konkretna posiada własną tabelę w bazie danych zawierającą atrybuty specyficzne dla swojej kategorii. Strategia ta wynika z istotnego rozmiaru atrybutów *Client* (dane osobowe, dokument tożsamości, dane firmy) względem *Employee* i *Admin*; alternatywna strategia *Table-per-Hierarchy* generowałaby tabelę z dużą liczbą kolumn nullowalnych.

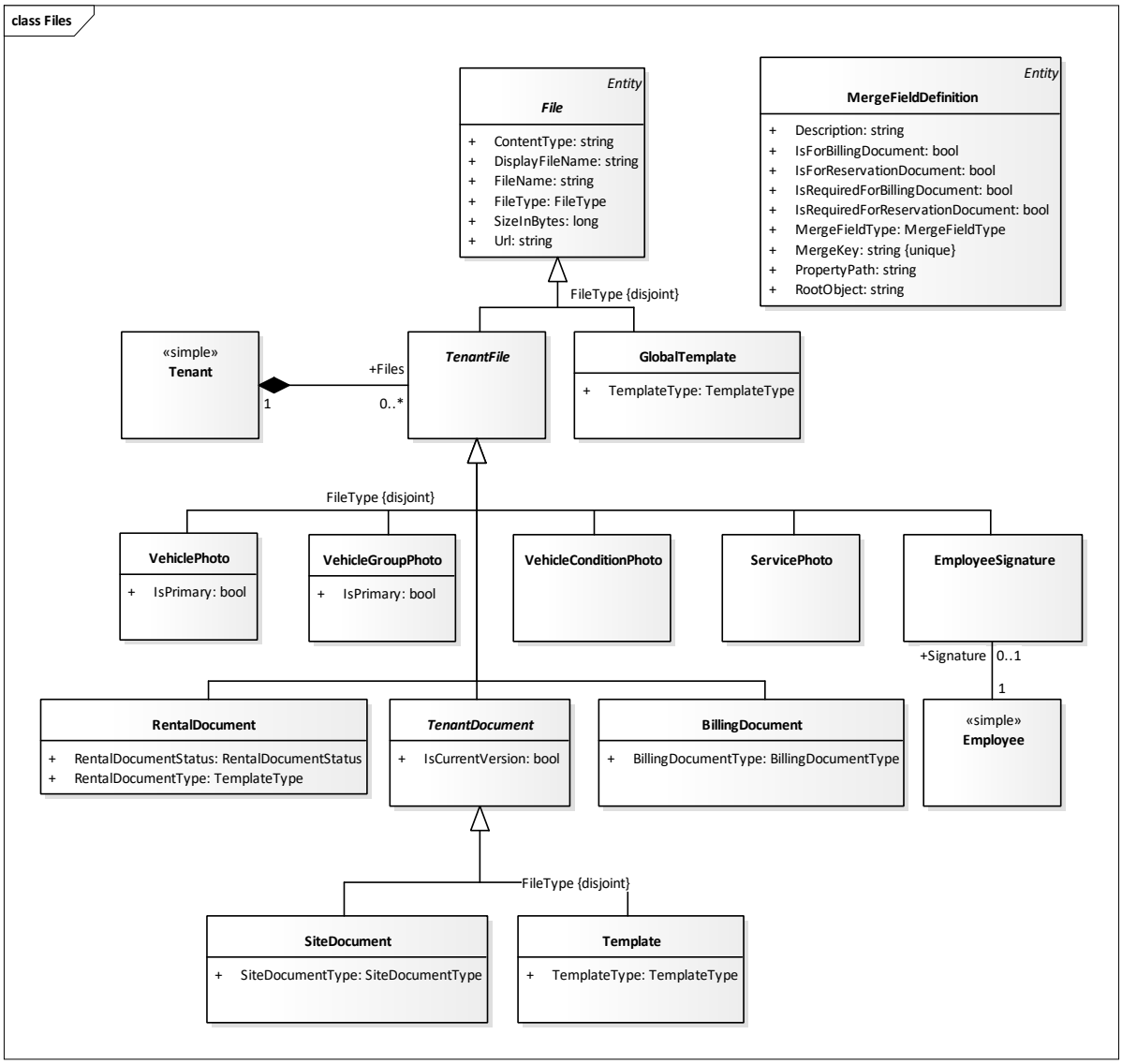
Ograniczenie widoczne obok klasy *User* definiuje zakres unikalności adresu e-mail w zależności od typu konta: dla klientów obowiązuje unikalność w obrębie wypożyczalni (*{unique per*

Tenant}), natomiast dla pracowników biurowych i administratorów platformy – unikalność globalna w skali całego systemu.

Wymagania funkcjonalne związane z zarządzaniem kontami opisano w punktach 5.3.3 oraz 5.3.11.

6.3.3.3. Files

Moduł Files przedstawiony na *Rysunek 62* prezentuje hierarchię plików i dokumentów przechowywanych w systemie. Abstrakcyjna klasa bazowa *File* zawiera wspólne metadane dla wszystkich plików i ma dwie bezpośrednie klasy potomne, realizujące podstawowy podział zakresu: *GlobalTemplate* reprezentuje globalny szablon platformy, natomiast abstrakcyjna klasa *TenantFile* wprowadza powiązanie z konkretnym tenantem dla wszystkich pozostałych plików. Z *TenantFile* dziedziczą klasy konkretnych plików tenantowych (*BillingDocument*, *EmployeeSignature*, *RentalDocument*, *ServicePhoto*, *VehicleConditionPhoto*, *VehicleGroupPhoto*, *VehiclePhoto*) oraz pośrednia klasa *TenantDocument* (dodająca flagę *IsCurrentVersion* dla wersjonowania dokumentów), z której z kolei dziedziczą *Template* i *SiteDocument*.



Rysunek 62 PCD modułu Files. Źródło: opracowanie własne.

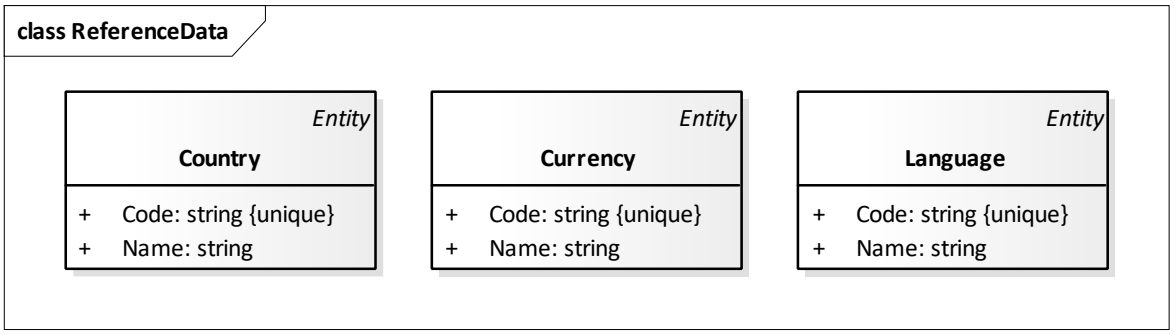
Cała hierarchia plików utrwalana jest strategią *Table-per-Hierarchy* – wszystkie typy plików znajdują się we wspólnej tabeli z dyskryminatorem *FileType*, który rozróżnia konkretny typ. Jednolitość schematu umożliwia efektywne zapytania o wszystkie pliki tenanta bez konieczności wykonywania złączeń między tabelami specjalizowanymi.

Na diagramie obecna jest również klasa *MergeFieldDefinition* (z unikalnym identyfikatorem *MergeKey*), wykorzystywana do scalania danych w szablonach dokumentów. Opis mechanizmu znajduje się w *7.1.10*.

Pliki są wykorzystywane przekrojowo przez wiele modułów funkcjonalnych – zdjęcia pojazdów i raportów stanu technicznego w punkcie *5.3.2*, dokumenty wynajmu oraz podpisy pracowników w punkcie *5.3.5*, a także dokumenty rozliczeniowe i szablony w punkcie *5.3.6* oraz dokumenty związane ze stroną publiczną platformy w punkcie *5.3.13*.

6.3.3.4. ReferenceData

Moduł ReferenceData przedstawiony na *Rysunek 63* prezentuje trzy słowniki referencyjne systemu – *Country* (kraje), *Currency* (waluty) i *Language* (języki). Wszystkie trzy są encjami posiadającymi własną tożsamość przez identyfikator i są referowane z innych agregatów przez powiązanie kluczem obcym, a nie przez kopiowanie wartości.

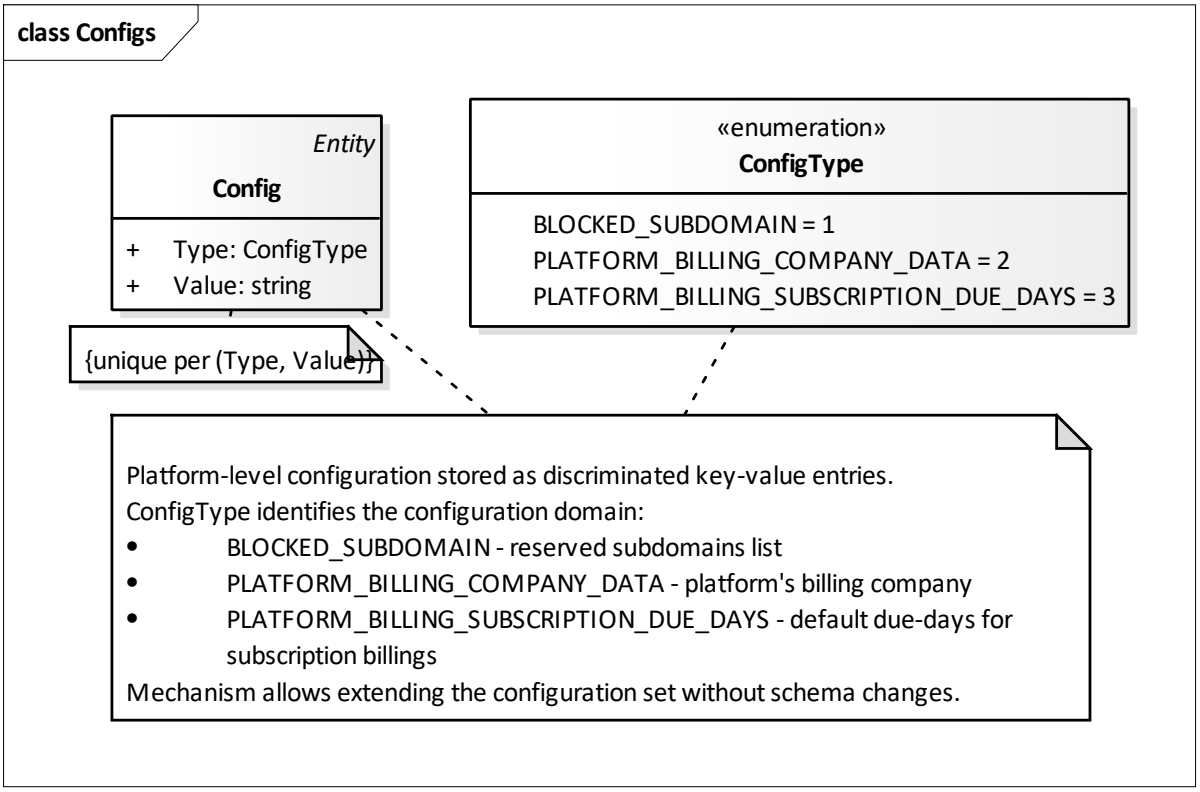


Rysunek 63 PCD modułu ReferenceData – słowniki referencyjne kraju, waluty i języka. Źródło: opracowanie własne.

Atrybuty *Country.Code*, *Currency.Code* i *Language.Code* posiadają ograniczenie *{unique}* – kody odpowiadają standardom międzynarodowym (ISO 3166-1 alpha-2 dla kraju, ISO 4217 dla waluty, ISO 639-1 dla języka) i stanowią naturalny klucz biznesowy zapewniający jednoznaczną identyfikację rekordów słownikowych. Klasy te nie implementują interfejsu *IAuditable*. Słowniki referencyjne nie odpowiadają osobnemu modułowi funkcjonalnemu.

6.3.3.5. Configs

Moduł Configs przedstawiony na *Rysunek 64* prezentuje pojedynczą klasę *Config* przechowującą konfigurację platformy w postaci dyskryminowanych par klucz-wartość. Każdy rekord jest identyfikowany przez typ konfiguracji (*ConfigType*) oraz wartość, której struktura zależy od typu (najczęściej tekst, identyfikator zasobu lub serializowany JSON).



Rysunek 64 PCD modułu Configs – konfiguracja platformy jako dyskryminowane pary klucz-wartość. Źródło: opracowanie własne.

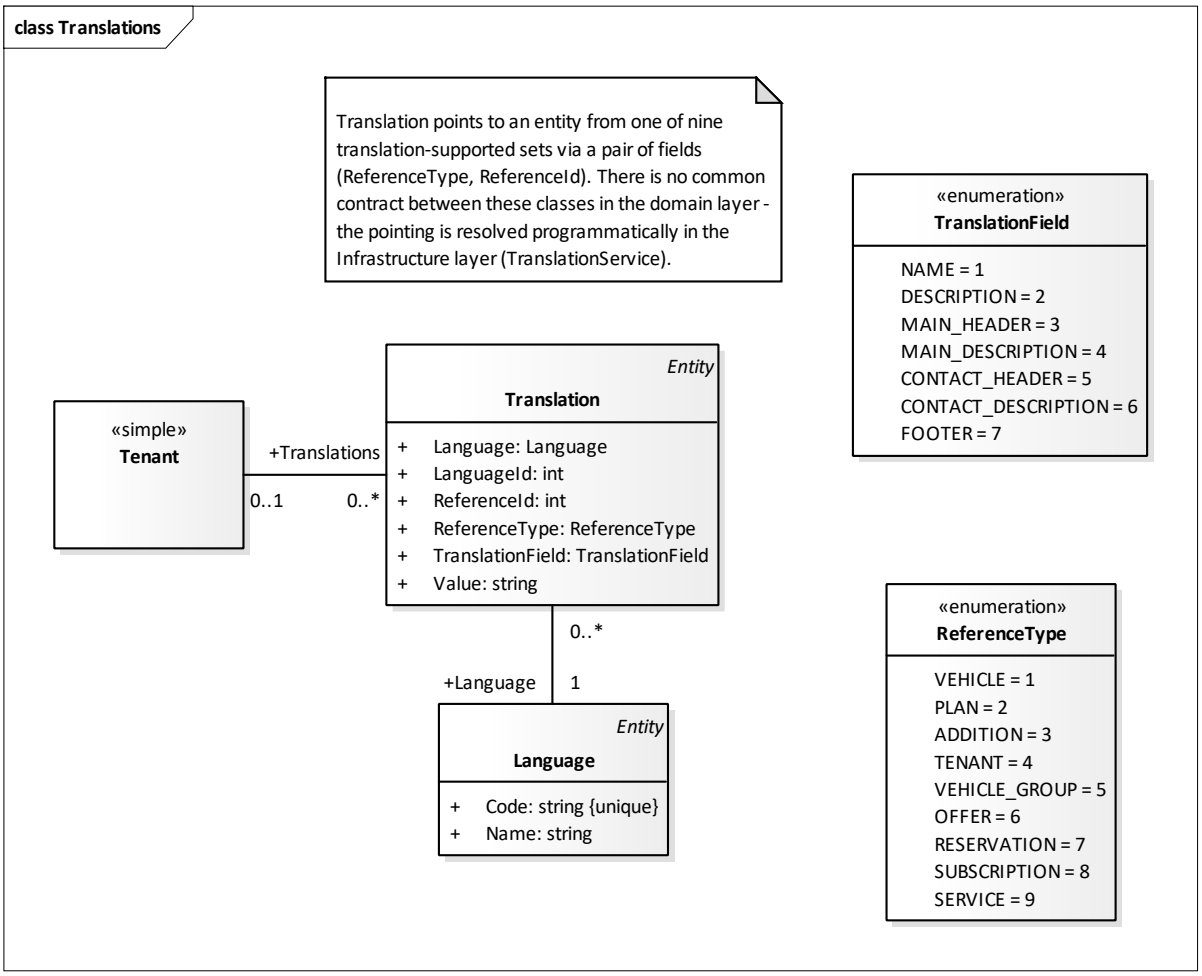
Konfiguracja ma zasięg globalny platformy – *Config* nie posiada powiązania z tenantem. Na diagramie zaznaczono ograniczenie *{unique per (Type, Value)}* – kombinacja typu i wartości musi być unikalna. Wzorzec ten pozwala na wiele wpisów tego samego typu (np. lista zarezerwowanych subdomen jako wiele wpisów *BLOCKED_SUBDOMAIN* z różnymi wartościami) bez ryzyka duplikacji.

Rozszerzalność konfiguracji realizowana jest przez dodawanie wartości do enumeracji *ConfigType* (punkt *6.3.4*, Enums-Tenants) – nie wymaga to tworzenia nowych tabel ani modyfikacji schematu bazy danych. Aktualnie zdefiniowane typy obejmują listę zablokowanych subdomen (*BLOCKED_SUBDOMAIN*), dane firmy używane w rozliczeniach platformy (*PLATFORM_BILLING_COMPANY_DATA*) oraz domyślny termin płatności dla rozliczeń subskrypcji (*PLATFORM_BILLING_SUBSCRIPTION_DUE_DAYS*).

Wymagania funkcjonalne związane z konfiguracją globalną opisane są w punkcie *5.3.11*.

6.3.3.6. Translations

Moduł Translations przedstawiony na *Rysunek 65* prezentuje polimorficzny mechanizm tłumaczeń wieloznacznych pól tekstowych w systemie. Pojedyncza encja *Translation* przechowuje wartość tłumaczenia, język, którego ono dotyczy, oraz parę identyfikującą encję źródłową: typ (*ReferenceType*) i identyfikator (*ReferenceId*). Dziewięć typów encji domenowych podlega tłumaczeniu – pojazdy, plany subskrypcyjne, dodatki, tenanci, grupy pojazdów, oferty, rezerwacje, subskrypcje oraz serwisy.



Rysunek 65 PCD modułu Translations – polimorficzny mechanizm tłumaczeń wielu typów encji. Źródło: opracowanie własne.

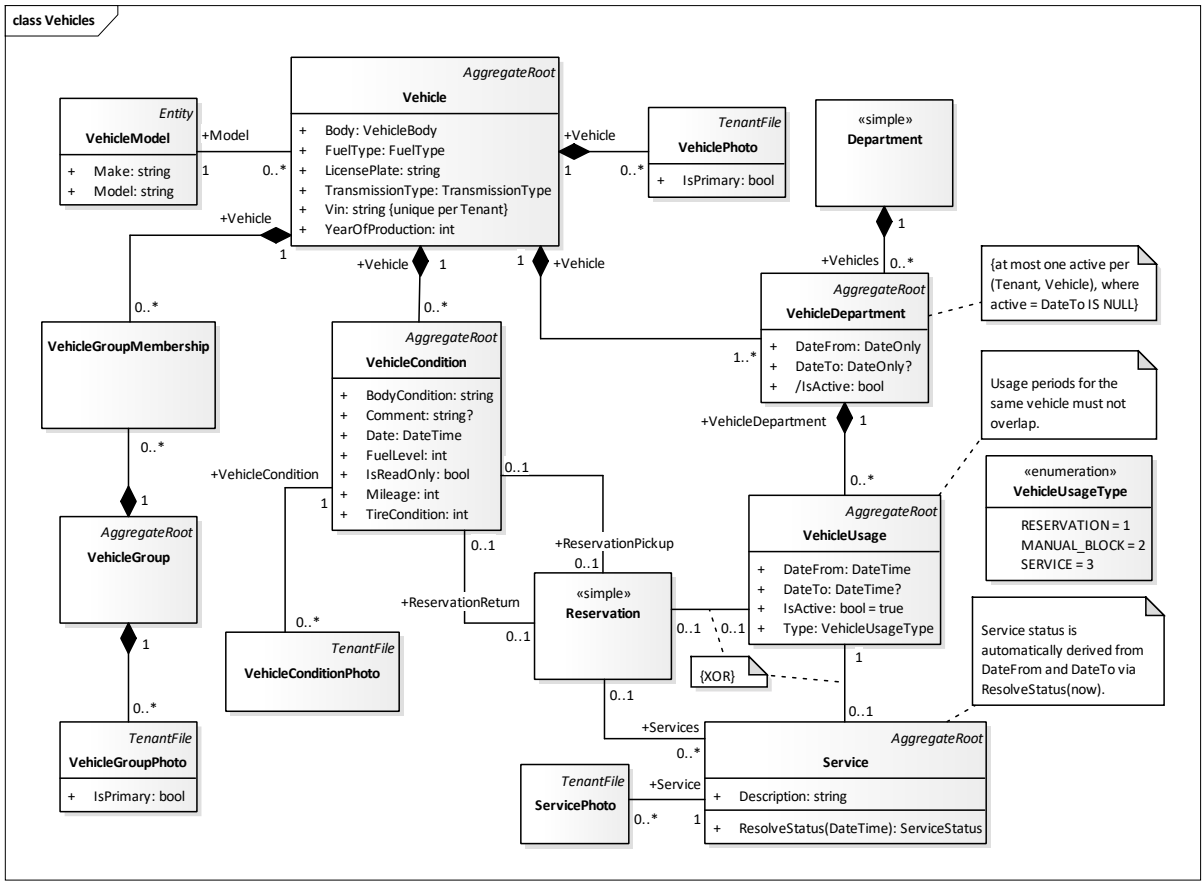
Na diagramie nie pokazano klasycznej relacji UML pomiędzy *Translation* a tłumaczonymi encjami – związek jest polimorficzny i rozwiązywany poprzez serwis w warstwie infrastruktury, bez wspólnego kontraktu po stronie domeny. Notatka przy klasie objaśnia mechanizm.

Tłumaczenia dzielą się na globalne (np. nazwy pakietu subskrypcyjnego) oraz lokalne dla konkretnego tenanta – tłumaczenia lokalne nadpisują tłumaczenia globalne tej samej pary (*ReferenceType*, *ReferenceId*, *TranslationField*). Atrybut *ReferenceType* jest dyskryminatorem typu encji źródłowej, a atrybut *TranslationField* – dyskryminatorem tłumaczonego pola (nazwa, opis, nagłówek, treść kontaktu, stopka).

Mechanizm tłumaczeń stosowany jest przekrojowo przez większość modułów funkcjonalnych – wszędzie tam, gdzie publiczne nazwy i opisy mogą być wyświetlane klientom wypożyczalni w różnych językach.

6.3.3.7. Vehicles

Moduł Vehicles przedstawiony na *Rysunek 66* prezentuje agregat pojazdu (*Vehicle*) wraz z encjami opisującymi jego cykl użytkowania – grupy pojazdów (*VehicleGroup*), przypisanie do oddziałów w czasie (*VehicleDepartment*), rejestr użyć (*VehicleUsage*), raporty stanu technicznego (*VehicleCondition*) oraz serwisy (*Service*). *VehicleModel* reprezentuje markę i model jako encję referencyjną wspólną dla wielu egzemplarzy pojazdów.



Rysunek 66 PCD modułu Vehicles – agregat pojazdu i powiązane encje cyklu użytkowania. Źródło: opracowanie własne.

Atrybut *Vehicle.Vin* posiada ograniczenie *{unique per Tenant}* – numer VIN jest unikalny w obrębie floty pojedynczego tenanta, ale ten sam VIN może wystąpić w innym tenancie.

Encja *VehicleDepartment* reprezentuje przypisanie pojazdu do oddziału. Atrybut *DateFrom* wskazuje początek przypisania, natomiast *DateTo* typu *DateOnly?* rozróżnia dwa rodzaje rekordów: przypisanie bezterminowo otwarte (*DateTo = null*) oraz przypisanie z zaplanowanym lub historycznym zakończeniem (*DateTo* ustawione).

Klasa *VehicleUsage* rejestruje każde użycie pojazdu – rezerwację, blokadę ręczną lub serwis (dyskryminowane przez *VehicleUsageType*). Notatka przy klasie informuje o niezmienniku rozłączności czasowej: okresy użyć tego samego pojazdu nie mogą się nakładać.

Powiązanie *Reservation-VehicleUsage* ma krotność 0..1-0..1: rezerwacja wskazuje co najwyżej jedno użycie pojazdu, a użycie może być blokadą lub serwisem niezwiązanym z rezerwacją. Zmiana pojazdu w trakcie rezerwacji zastępuje dotychczasowe użycie, dlatego historia wcześniejszych przypisań pozostaje świadomie poza zakresem modelu.

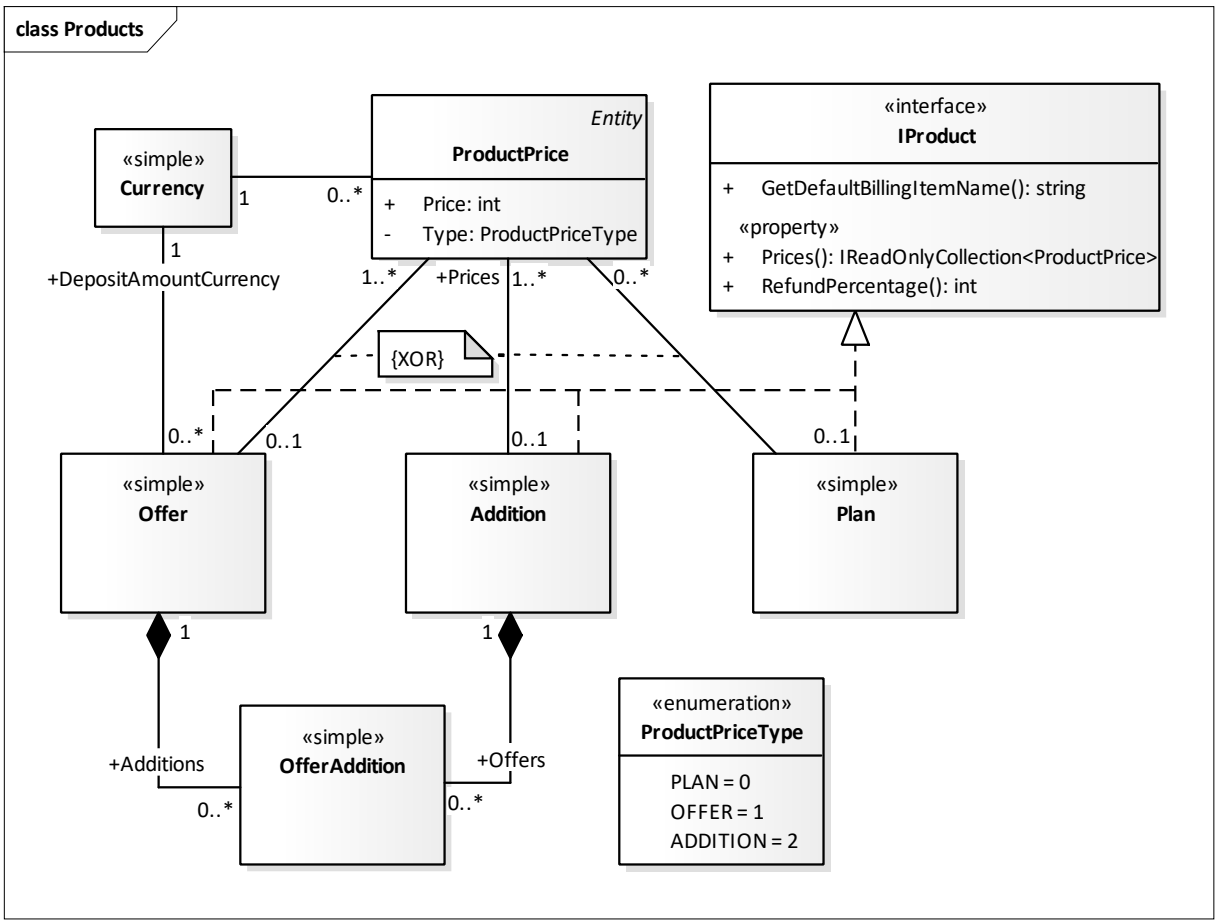
Status serwisu (*Service.Status*) jest wyznaczany dynamicznie – wartość typu *ServiceStatus* obliczana jest na podstawie atrybutów *DateFrom* i *DateTo* w odniesieniu do bieżącego momentu.

Wymagania funkcjonalne związane z zarządzaniem flotą opisano w punkcie *5.3.2*, a procedurę zgłaszania usterki przez pracownika operacyjnego przedstawiono w punkcie *5.3.8*.

6.3.3.8. Products

Moduł Products przedstawiony na *Rysunek 67* prezentuje interfejs *IProduct* jako wspólny kontrakt dla produktów oferowanych w systemie oraz trzy klasy implementujące – *Offer* (oferta

wynajmu pojazdu lub grupy pojazdów), *Addition* (dodatek do rezerwacji) i *Plan* (plan subskrypcyjny platformy). Każdy produkt posiada cennik reprezentowany przez kolekcję *ProductPrice* zawierającą wartości w różnych walutach.



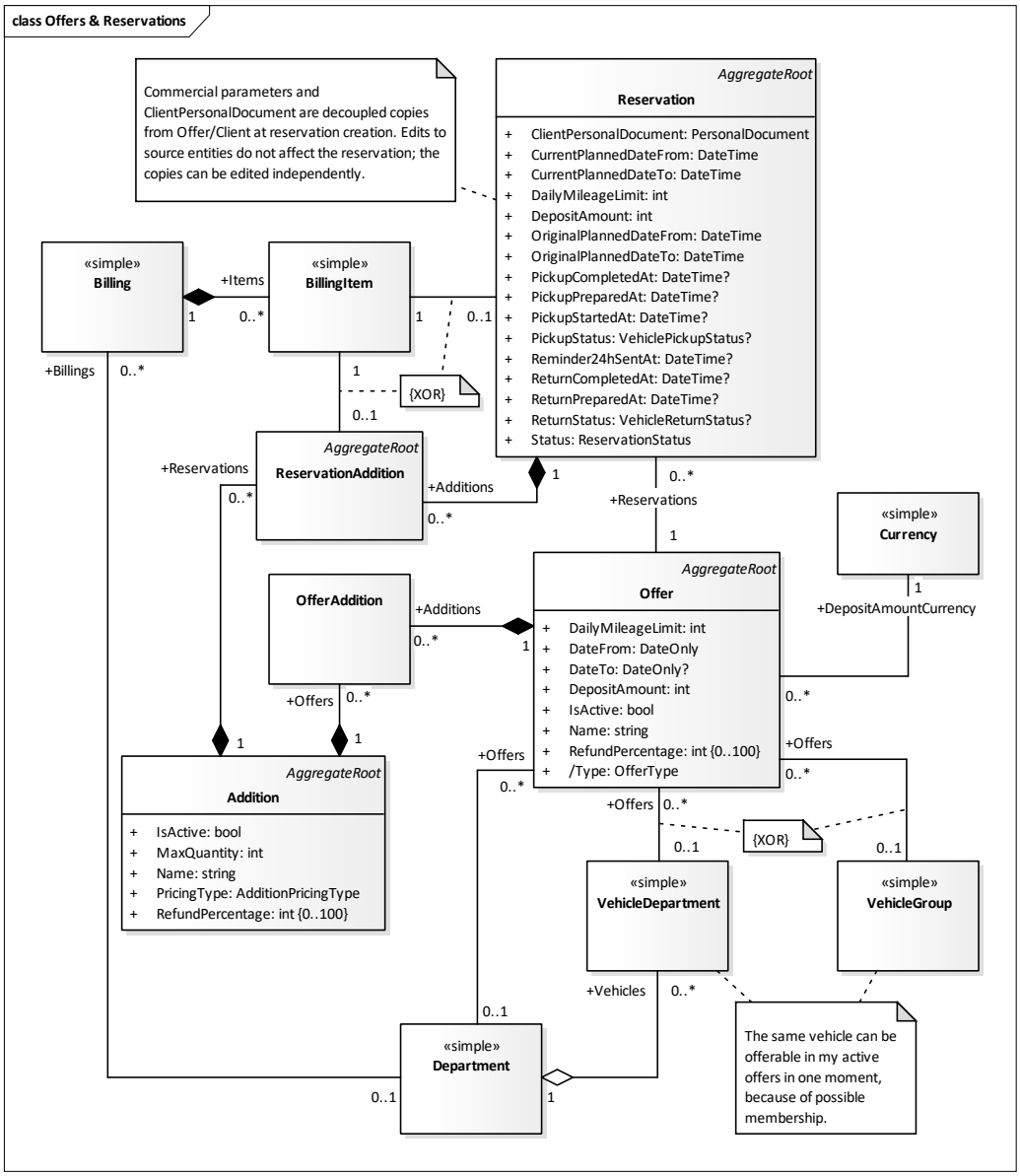
Rysunek 67 PCD modułu Products – interfejs IProduct i klasy implementujące. Źródło: opracowanie własne.

Klasa *ProductPrice* wskazuje na konkretny produkt przez relację XOR – dokładnie jedno powiązanie z *Plan*, *Offer* lub *Addition* jest aktywne. Dyskryminator typu produktu reprezentowany jest przez atrybut *Type* typu *ProductPriceType* (punkt *6.3.4*, Enums-Finances).

Wymagania funkcjonalne dotyczące ofert wynajmu zostały opisane w punkcie *5.3.4*, natomiast plany subskrypcyjne omówiono w punkcie *5.3.10*. Pełne definicje klas implementujących interfejs *IProduct*, tj. *Offer* oraz *Addition* i *Plan*, znajdują się odpowiednio w podpunktach *6.3.3.9* i *6.3.3.10*.

6.3.3.9. Offers & Reservations

Offers & Reservations przedstawione na *Rysunek 68* prezentuje ofertę (*Offer*) wraz z jej realizacją w postaci rezerwacji (*Reservation*), a także dodatki dostępne dla ofert (*Addition*) i ich konkretne wystąpienia w rezerwacji (*ReservationAddition*). Klasa *OfferAddition* reprezentuje powiązanie, które określa, jakie dodatki są dostępne dla danej oferty.



Rysunek 68 PCD modułu Offers & Reservations – oferty, rezerwacje i dodatki. Źródło: opracowanie własne.

Offer jest powiązany z jednym z dwóch celów oferty: konkretnym pojazdem w oddziale (przez *VehicleDepartment*) lub grupą pojazdów (*VehicleGroup*). Atrybuty *Offer.RefundPercentage* oraz *Addition.RefundPercentage* z ograniczeniem *{0..100}* definiują procent zwrotu wykorzystywany przy anulowaniu rezerwacji lub dodatku.

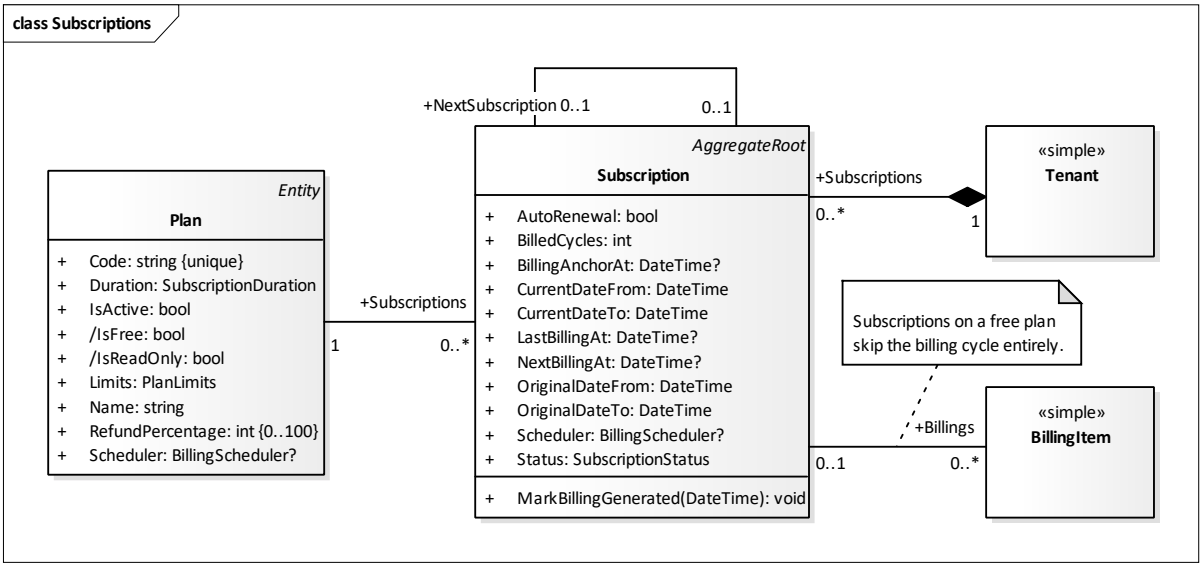
Rezerwacja przechowuje dwie pary atrybutów dat planowanych: oryginalną (*OriginalPlannedDateFrom*, *OriginalPlannedDateTo*) zapisaną w momencie utworzenia rezerwacji oraz bieżącą (*CurrentPlannedDateFrom*, *CurrentPlannedDateTo*) aktualizowaną przy każdej modyfikacji harmonogramu. Różnica między obiema parami umożliwia śledzenie historii zmian dat rezerwacji w trakcie jej obsługi.

Notatka przy klasie *Reservation* informuje o wzorcu kopii – parametry handlowe (cena, depozyt, liczba dni) oraz dane dokumentu osobistego klienta są zapisywane jako kopie w momencie tworzenia rezerwacji. Modyfikacja źródłowej *Offer* lub *Client* nie propaguje się do utworzonej rezerwacji, natomiast same kopie pozostają edytowalne z poziomu rezerwacji bez wpływu na encje źródłowe.

Wymagania funkcjonalne związane z ofertami opisane są w punkcie *5.3.4*, a dotyczące rezerwacji – w punkcie *5.3.5*.

6.3.3.10. Subscriptions

Moduł Subscriptions przedstawiony na *Rysunek 69* prezentuje subskrypcje SaaS klientów platformy (tenantów). *Subscription* jest agregatem reprezentującym aktywne lub historyczne korzystanie z planu, *Plan* – produkt subskrypcyjny z cennikiem.



Rysunek 69 PCD modułu Subscriptions – subskrypcje i plany cenowe. Źródło: opracowanie własne.

Atrybut *Plan.Code* z ograniczeniem *{unique}* stanowi naturalny klucz biznesowy, umożliwiający jednoznaczne odwołanie do planu spoza systemu (np. w integracjach z operatorem płatności). Atrybut *Plan.RefundPercentage* z ograniczeniem *{0..100}* określa domyślny procent zwrotu stosowany przy rezygnacji z subskrypcji – wartość ta jest wykorzystywana do obliczania kwoty refundacji w module Payments.

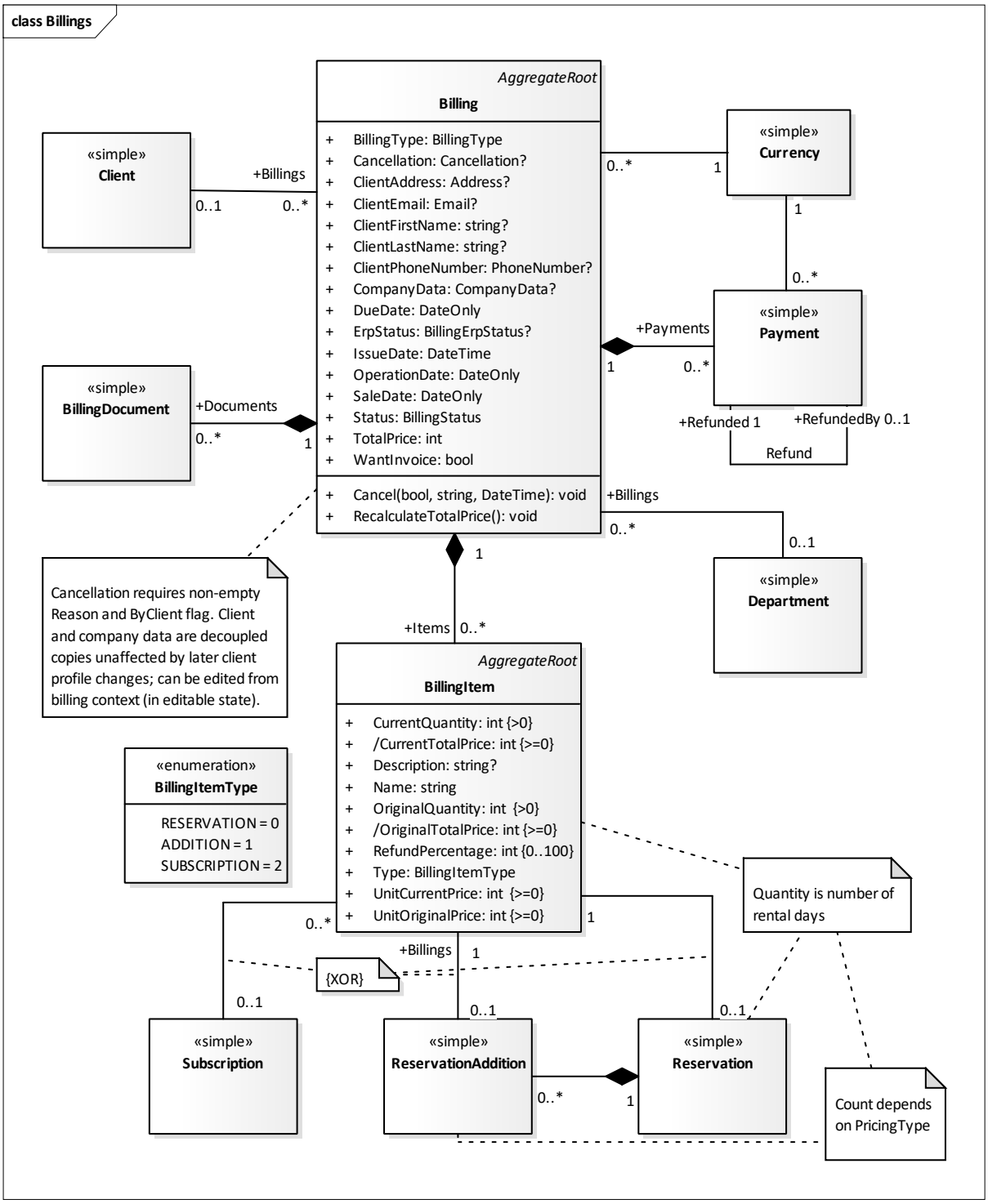
Notatka przy klasie Subscription informuje, że subskrypcje na planie darmowym (*Plan.IsFree*) pomijają cykl rozliczeniowy – nie generują dokumentów rozliczeniowych ani powiązanych płatności.

Relacja *Subscription -> NextSubscription* wskazuje subskrypcję, która ma zastąpić bieżącą. Mechanizm nie modeluje pełnej historii subskrypcji dla konkretnego tenanta.

Wymagania funkcjonalne związane z subskrypcjami i planami opisane są w punkcie *5.3.10*.

6.3.3.11. Billings

Moduł Billings przedstawiony na *Rysunek 70* prezentuje dokumenty rozliczeniowe (*Billing*) wraz z ich pozycjami (*BillingItem*). Agregat *Billing* reprezentuje pojedynczy dokument rozliczeniowy – dokument sprzedaży lub zakupu.



Rysunek 70 PCD modułu Billings – dokumenty rozliczeniowe i ich pozycje. Źródło: opracowanie własne.

Atrybut *Billing.BillingType* (punkt *6.3.4,* Enums-Finances) rozróżnia dwa kierunki dokumentu: *SALE* (sprzedaż realizowana przez tenanta – rezerwacja, dodatek) oraz *PURCHASE* (subskrypcja platformy Wellhicle kupowana przez tenanta – dokument zakupu od dostawcy platformy).

Atrybut *Billing.Status* przyjmuje cztery wartości: *PENDING* (oczekuje rozliczenia), *PAID* (opłacony), *COMPLETED* (zakończony) oraz *CANCELLED* (anulowany). Statusy *COMPLETED* i *CANCELLED* czynią dokument tylko do odczytu. Status *BillingErpStatus* reprezentuje niezależny cykl synchronizacji dokumentu z systemem ERP.

Notatka przy klasie *Billing* informuje o wzorcu kopii – pola *ClientFirstName*, *ClientLastName*, *ClientEmail*, *ClientPhoneNumber*, *ClientAddress* oraz *CompanyData* są snapshotem wykonanym w momencie wystawienia dokumentu. Modyfikacje danych klienta lub tenanta po wystawieniu nie propagują się do dokumentu, natomiast kopie pozostają edytowalne z poziomu kontekstu dokumentu (gdy znajduje się w stanie umożliwiającym modyfikację).

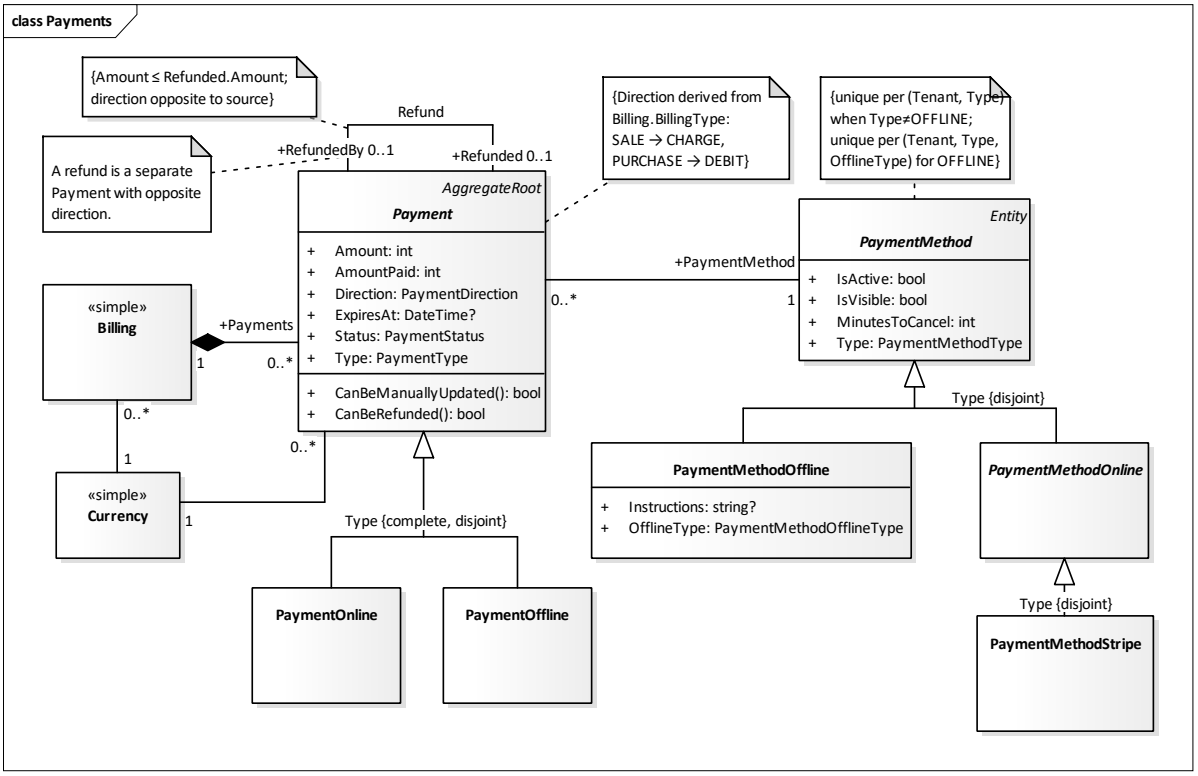
BillingItem posiada trzy alternatywne referencje wskazujące na encję biznesową stanowiącą podstawę pozycji: *Reservation*, *Addition* lub *Subscription*. Dokładnie jedna z nich jest ustawiona; relacja oznaczona stereotypem *XOR*. Dyskryminator typu reprezentowany jest przez *BillingItemType*.

Atrybuty ilościowe *BillingItem.OriginalQuantity* i *CurrentQuantity* posiadają ograniczenie *{>0}* – ilość rozliczeniowa musi być zawsze dodatnia. Atrybuty kwotowe *UnitOriginalPrice*, *UnitCurrentPrice*, *OriginalTotalPrice*, *CurrentTotalPrice* mają ograniczenie *{>=0}* – wartości wyrażone są w minimalnych jednostkach waluty (np. groszach), przy czym nie dopuszcza się kwot ujemnych.

Wymagania funkcjonalne związane z rozliczeniami opisane są w punkcie *5.3.6*.

6.3.3.12. Payments

Moduł Payments przedstawiony na *Rysunek 71* prezentuje płatności i metody płatności. Abstrakcyjna klasa *Payment* ma dwie konkretne klasy potomne: *PaymentOnline* (płatność online realizowana przez operatora) oraz *PaymentOffline* (płatność tradycyjna). Hierarchia metod jest dwuwarstwowa – abstrakcyjna klasa *PaymentMethod* dziedziczona jest przez *PaymentMethodOffline* (obejmującą płatności gotówkowe, przelewy, czeki, kryptowaluty i inne) oraz przez pośrednią abstrakcyjną klasę *PaymentMethodOnline*, z której dziedziczy *PaymentMethodStripe*. Taka konstrukcja umożliwia dodawanie kolejnych operatorów płatności online (np. PayU czy Przelewy24) bez modyfikacji hierarchii metod offline.



Rysunek 71 PCD modułu Payments – płatności i metody płatności. Źródło: opracowanie własne.

Obie hierarchie utrwalane są strategią *Table-per-Hierarchy – Payment* z dyskryminatorem *Type* typu *PaymentType* (*ONLINE*/*OFFLINE*) oraz *PaymentMethod* z dyskryminatorem *Type* typu *PaymentMethodType* (*STRIPE*/*OFFLINE*). Dla metod offline dodatkowe rozróżnienie podtypu (gotówka, przelew, czek, kryptowaluta, inne) realizowane jest atrybutem *OfflineType*; *PaymentMethodOfflineType* na klasie *PaymentMethodOffline*.

Constraint przy klasie Payment definiuje kierunek płatności (*Direction*) jako wyznaczany z atrybutu *BillingType* powiązanego dokumentu rozliczeniowego: *SALE -> CHARGE* (środki wpływające do tenanta), *PURCHASE -> DEBIT* (środki wypływające).

Notatka na diagramie precyzuje wzorzec realizacji zwrotów: zwrot środków stanowi osobną instancję klasy Payment o kierunku przeciwnym do płatności pierwotnej. Kwota zwrotu nie może przekraczać kwoty oryginalnej płatności.

Unikalność metod płatności w obrębie tenanta jest warunkowa: tenant może posiadać dokładnie jedną metodę typu Stripe oraz wiele metod offline (po jednej każdego podtypu).

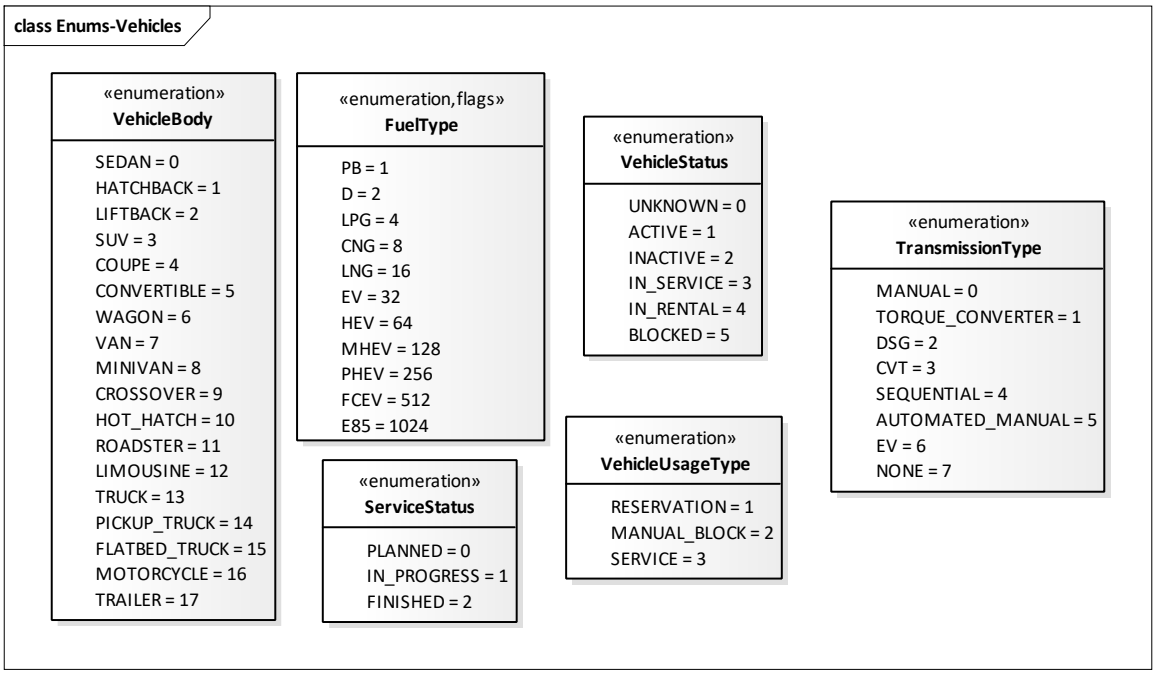
Wymagania funkcjonalne związane z płatnościami opisane są w punkcie *5.3.7*.

6.3.4. Enumeracje

Punkt zawiera dziewięć diagramów przedstawiających katalogi typów wyliczeniowych systemu Wellhicle, pogrupowane tematycznie. Wyodrębnienie enumeracji na osobne diagramy wynika z ich liczebności – niektóre wymieniają kilkanaście wartości (np. *VehicleBody* ma 18, *FuelType* 11, *EmailType* 10). Powtarzanie pełnych katalogów na PCD modułowych z sekcji *6.3.3* utrudniłoby analizę relacji wewnątrz modułu. Na diagramach modułowych pozostałe typy wyliczeniowe występują jako typy atrybutów, natomiast typy kluczowe dla danego modułu (dyskryminatory) pokazano dodatkowo jako osobne elementy <<*enumeration*>> przy odpowiednich klasach.

6.3.4.1. Vehicles

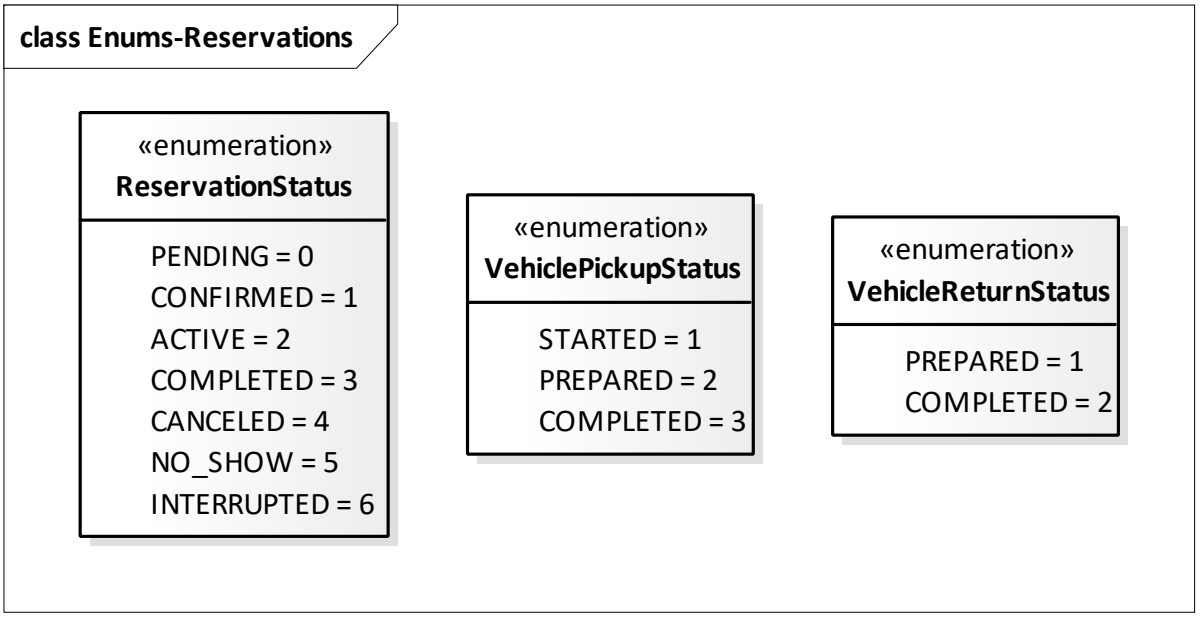
Diagram na *Rysunek 72* zawiera enumeracje opisujące pojazd i jego cykl użytkowania: *VehicleStatus* (wartości operacyjne – nieznany, aktywny, nieaktywny, w serwisie, w wynajmie, zablokowany), *VehicleBody* (typy nadwozia – od klasycznego sedana po naczepę), *FuelType* (rodzaje napędu, w tym kombinacje hybrydowe), *TransmissionType* (typy skrzyni biegów), *ServiceStatus* (stany serwisu – zaplanowany, w trakcie, zakończony) oraz *VehicleUsageType* (typy wykorzystania – rezerwacja, blokada ręczna, serwis).



Rysunek 72 Enumeracje modułu Vehicles. Źródło: opracowanie własne.

6.3.4.2. Reservations

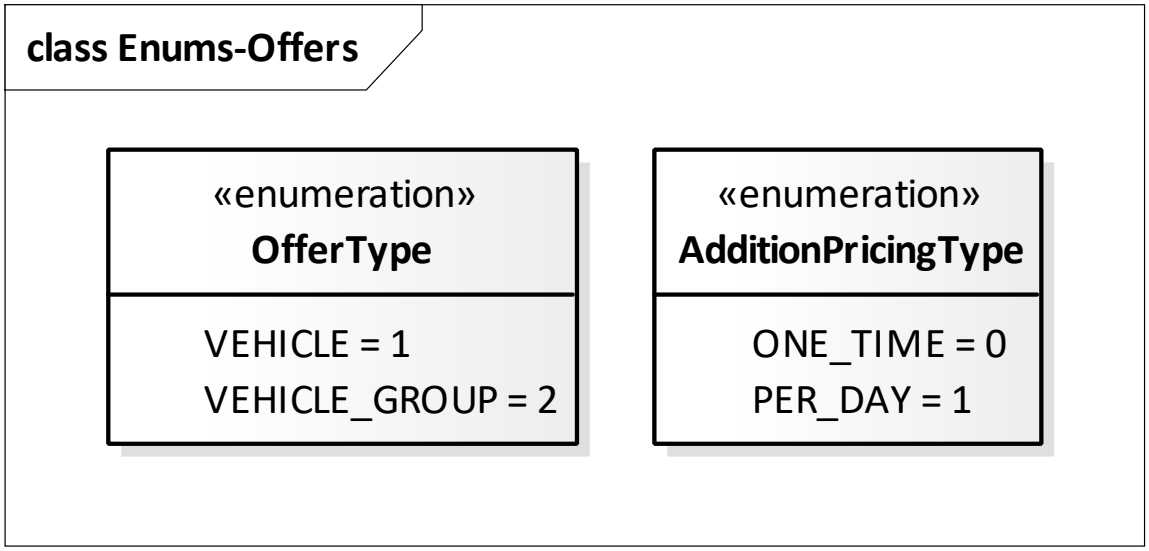
Diagram na *Rysunek 73* zawiera enumeracje opisujące stany rezerwacji i jej etapy realizacji: *ReservationStatus* (cykl życia rezerwacji – oczekująca, potwierdzona, aktywna, zakończona, anulowana, brak stawienia się, przerwana), *VehiclePickupStatus* (etapy wydania pojazdu – rozpoczęte, przygotowane, zakończone) oraz *VehicleReturnStatus* (etapy zwrotu pojazdu – przygotowany, zakończony).



Rysunek 73 Enumeracje modułu Reservations. Źródło: opracowanie własne.

6.3.4.3. Offers

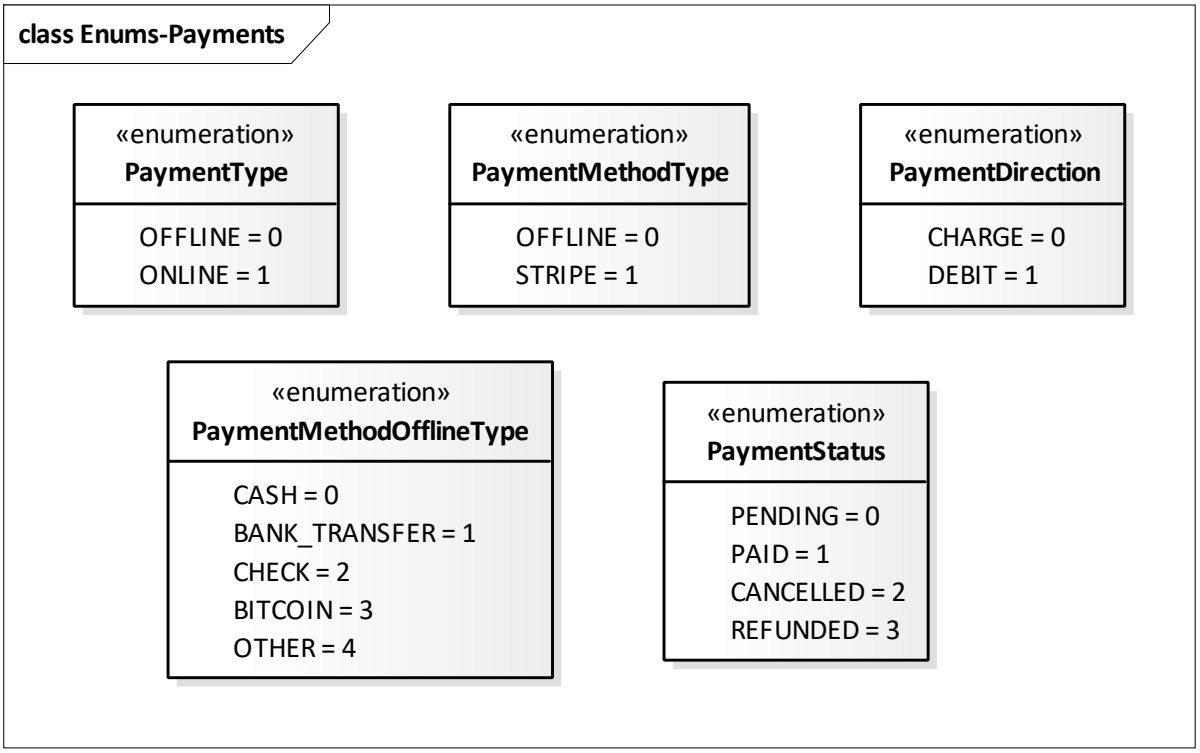
Diagram na *Rysunek 74* zawiera dwie enumeracje: *OfferType* (rozróżnienie ofert na pojedynczy pojazd lub grupę pojazdów – wartość wyznaczana przez obecność referencji *VehicleDepartment* lub *VehicleGroup*, patrz podpunkt *6.3.3.9*) oraz *AdditionPricingType* (model cenowy dodatku: jednorazowy lub za dzień).



Rysunek 74 Enumeracje modułu Offers. Źródło: opracowanie własne.

6.3.4.4. Payments

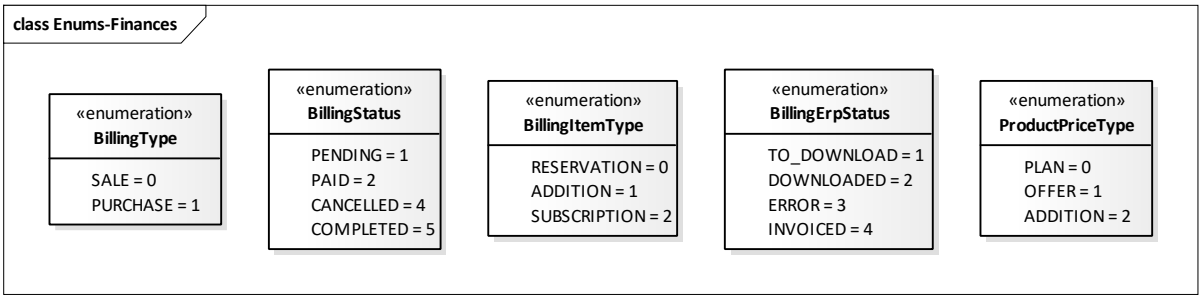
Diagram na *Rysunek 75* zawiera pięć enumeracji opisujących płatności i metody płatności: *PaymentStatus* (oczekująca, opłacona, anulowana, zwrócona), *PaymentDirection* (*CHARGE* – środki przychodzące, *DEBIT* – wychodzące), *PaymentType* (online lub offline), *PaymentMethodType* (dyskryminator polimorfizmu metod – *STRIPE* lub *OFFLINE*) oraz *PaymentMethodOfflineType* (gotówka, przelew bankowy, czek, Bitcoin, inne).



Rysunek 75 Enumeracje modułu Payments. Źródło: opracowanie własne.

6.3.4.5. Finances (rozliczenia)

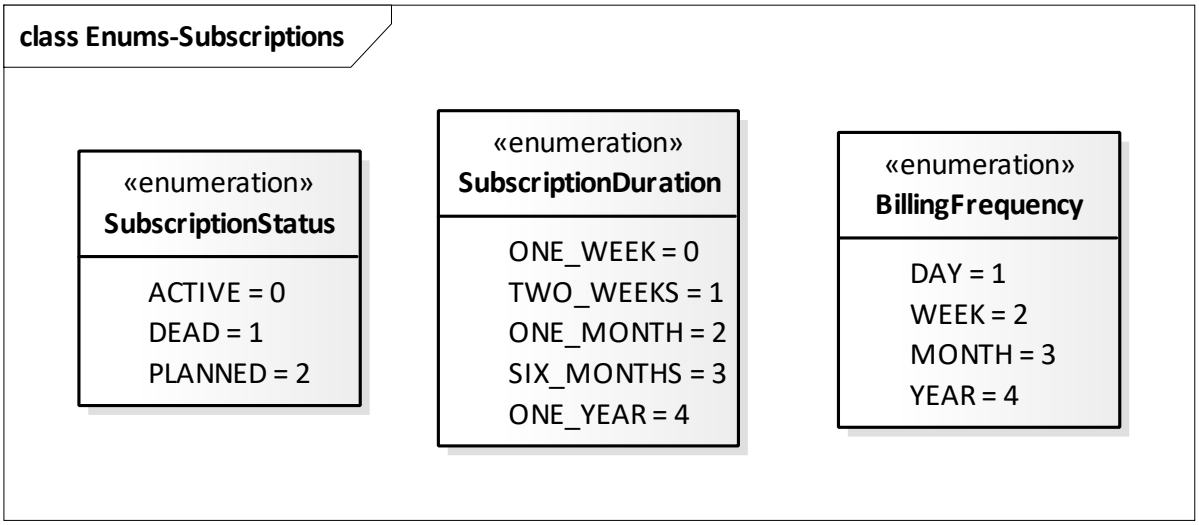
Diagram na *Rysunek 76* zawiera pięć enumeracji modułu rozliczeniowego: *BillingStatus* (oczekujący, opłacony, anulowany, zakończony – dwa ostatnie statusy czynią dokument tylko do odczytu), *BillingType* (*SALE* lub *PURCHASE* – określa kierunek dokumentu), *BillingItemType* (rezerwacja, dodatek, subskrypcja – rodzaj pozycji rozliczeniowej), *BillingErpStatus* (status synchronizacji z systemem ERP – do pobrania, pobrany, błąd, zafakturowany) oraz *ProductPriceType* (*PLAN*, *OFFER* lub *ADDITION* – typ produktu powiązanego z ceną; szczegóły w podpunkcie *6.3.3.8*).



Rysunek 76 Enumeracje modułu Finances (rozliczenia). Źródło: opracowanie własne.

6.3.4.6. Subscriptions

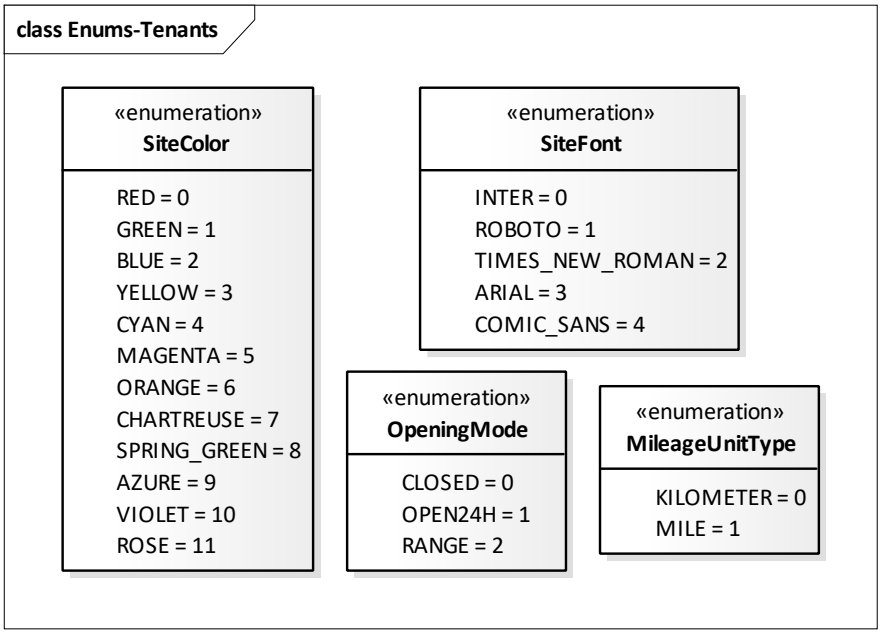
Diagram na *Rysunek 77* zawiera trzy enumeracje: *SubscriptionStatus* (aktywna, zakończona, zaplanowana), *SubscriptionDuration* (predefiniowane okresy subskrypcji – tydzień, dwa tygodnie, miesiąc, sześć miesięcy, rok) oraz *BillingFrequency* (jednostka cyklu rozliczeniowego – dzień, tydzień, miesiąc, rok – używana w *BillingScheduler*).



Rysunek 77 Enumeracje modułu Subscriptions. Źródło: opracowanie własne.

6.3.4.7. Tenants

Diagram na *Rysunek 78* zawiera enumeracje opisujące konfigurację oddziałów i witryny tenanta: *OpeningMode* (model godzin pracy oddziału – zamknięty, otwarty 24h, w przedziale), *SiteColor* (paleta kolorów dla witryny tenanta), *SiteFont* (dostępne czcionki witryny) oraz *MileageUnitType* (jednostka przebiegu pojazdów – kilometry lub mile).

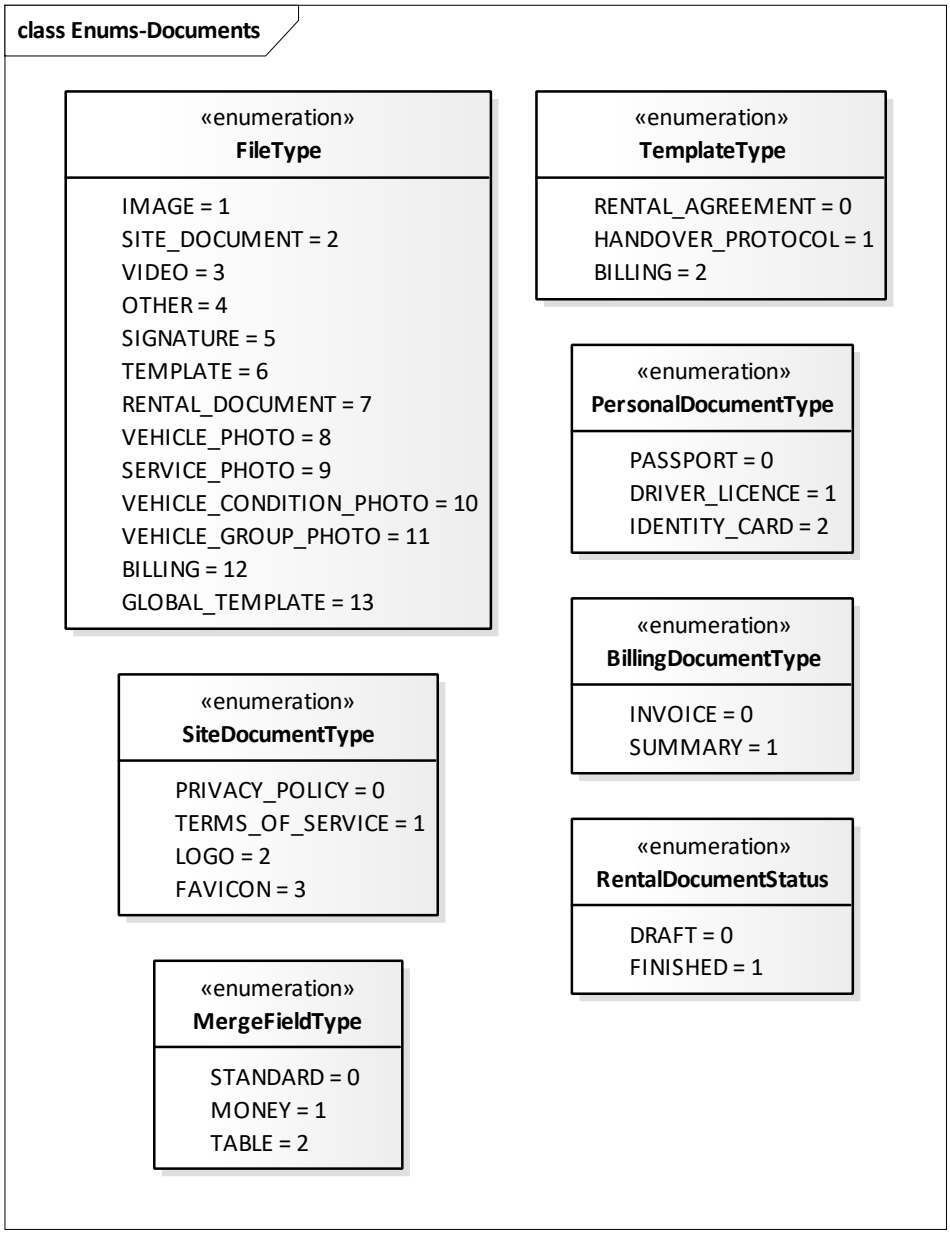


Rysunek 78 Enumeracje modułu Tenants. Źródło: opracowanie własne.

6.3.4.8. Documents

Diagram na *Rysunek 79* zawiera enumeracje opisujące typy plików i dokumentów: *FileType* (kategorie plików – od obrazów, przez dokumenty strony wypożyczalni oraz filmy lub podpisy elektroniczne po szablony czy zdjęcia floty; patrz podpunkt *6.3.3.3*), *BillingDocumentType* (faktura VAT albo zestawienie rozliczeniowe), *TemplateType* (umowa najmu, protokół wydania/zwrotu pojazdu lub dokument rozliczenia końcowego), *RentalDocumentStatus* (wersja robocza bądź ostateczna wraz

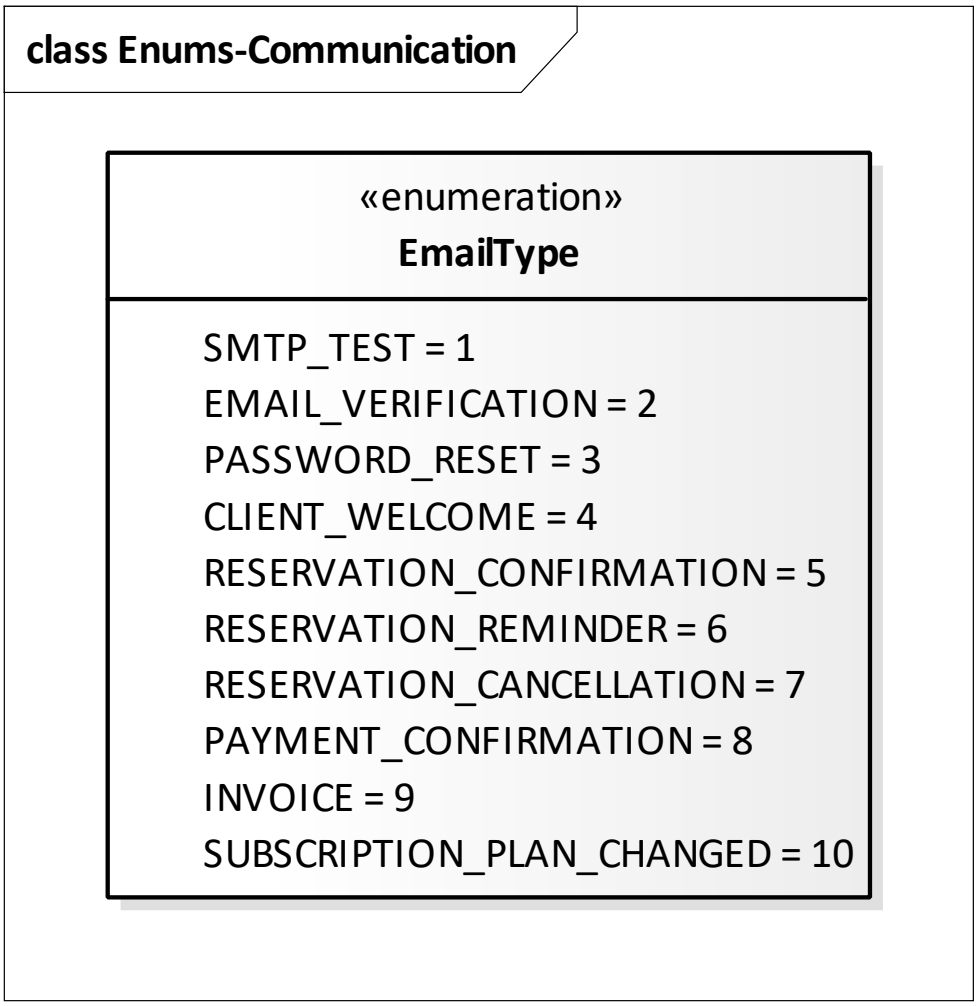
ze statusem zatwierdzenia przez platformę Wellhicle), *SiteDocumentType* (polityka prywatności, regulamin serwisu czy logo firmowe) oraz *MergeFieldType* (typ pola scalania danych – standardowy tekst, kwota finansowa lub tabelaryczne zestawienie). *PersonalDocumentType* obejmuje dokumenty tożsamości klientów: paszport, prawo jazdy bądź dowód osobisty.



Rysunek 79 Enumeracje typów plików i dokumentów. Źródło: opracowanie własne.

6.3.4.9. Communication

Diagram na *Rysunek 80* zawiera enumerację *EmailType* (typy wiadomości wysyłanych przez system – od testu konfiguracji SMTP i weryfikacji adresu, przez powitanie klienta i potwierdzenia rezerwacji, po fakturę oraz informację o zmianie planu subskrypcyjnego). Wybór wartości tej enumeracji determinuje wykorzystywaną konfigurację SMTP (systemową lub tenanta) oraz dostępne zmienne szablonu wiadomości.



Rysunek 80 Enumeracje modułu Communication. Źródło: opracowanie własne.

6.4. Projekt interfejsu użytkownika

Projekt interfejsu użytkownika obejmował prace związane z przygotowaniem warstwy wizualnej systemu Wellhicle. W tej części opisano wykorzystane narzędzia, elementy identyfikacji wizualnej oraz wybrane widoki opracowane na potrzeby projektu.

6.4.1. Nazwa i logo systemu

W ramach prac projektowych przygotowano nazwę systemu oraz kilka wariantów logo. Dzięki temu projekt interfejsu został uzupełniony o podstawowe elementy identyfikacji wizualnej.

Nazwa Wellhicle powstała z połączenia słów *well* (ang. „dobrze", „właściwie") oraz *vehicle* (ang. „pojazd"). Wskazuje ona na związek systemu z obsługą pojazdów, a jednocześnie ma zwięzłą i łatwą do zapamiętania formę.

Uzupełnieniem nazwy było opracowanie logo systemu oraz jego wariantów przedstawionych na *Rysunek 81*, przygotowanych z wykorzystaniem programu Inkscape. Element ten uzupełnia projekt interfejsu i porządkuje wizualną identyfikację systemu. Logo oparto na uproszczonym znaku typograficznym, którego głównym elementem jest litera „W", nawiązująca do nazwy Wellhicle. Znak uzupełniono o symbole lokalizacji, wskazujące na związek systemu z pojazdami i punktami ich

udostępniania. Całość zachowano w prostej, zwartej formie, odpowiedniej do wykorzystania w interfejsie systemu oraz materiałach projektowych.



Rysunek 81 Wersje logo przygotowane dla systemu Wellhicle. Źródło: opracowanie własne.

6.4.2. Założenia projektowe interfejsu

Projekt interfejsu użytkownika oparto na założeniu, że warstwa wizualna systemu powinna wspierać realizację zadań użytkownika oraz porządkować sposób korzystania z poszczególnych funkcji. Z tego względu przyjęto, że interfejs powinien być czytelny, spójny i przewidywalny, a jego struktura powinna ułatwiać orientację w systemie oraz wykonywanie najważniejszych operacji.

Jednym z podstawowych założeń było wykorzystanie istniejącego systemu projektowego (Material Design 3) jako bazy do przygotowania interfejsu. Ułatwiło to uporządkowanie kolorystyki, typografii oraz sposobu prezentacji informacji, a zarazem nie ograniczało możliwości tworzenia własnych komponentów tam, gdzie było to potrzebne. Dzięki temu interfejs zachował spójny charakter w różnych częściach systemu.

Przyjęto również, że projekt powinien docelowo obejmować całość systemu, tak aby wszystkie jego obszary były tworzone według wspólnych zasad wizualnych i funkcjonalnych. Założenie to miało zapewnić spójność pomiędzy poszczególnymi widokami oraz ograniczyć ryzyko powstawania rozwiązań projektowanych w oderwaniu od wcześniej ustalonych reguł.

Istotnym założeniem było zachowanie zgodności pomiędzy projektem interfejsu a późniejszą implementacją, przy jednoczesnym pozostawieniu swobody w zakresie drobnych szczegółów wizualnych. Projekt miał przede wszystkim określać ogólny wygląd systemu, układ widoków oraz sposób prezentacji podstawowych funkcji, a nie precyzyjnie definiować każdy element kompozycji.

W rezultacie założenia projektowe interfejsu koncentrowały się na zapewnieniu spójności wizualnej, przewidywalności obsługi oraz możliwości późniejszego przełożenia projektu na implementację systemu.

6.4.3. Projekt wybranych widoków systemu

W procesie projektowania widoków wykorzystano powtarzalne komponenty interfejsu. Pozwoliło to zachować podobny sposób prezentacji i obsługi danych w różnych częściach systemu. Widoki listy, szczegółów oraz edycji różniły się głównie zakresem prezentowanych informacji, natomiast ich ogólny układ i sposób działania pozostały zbliżone.

Material Design 3 wykorzystano jako podstawę do zachowania spójności wizualnej komponentów oraz ujednolicenia takich elementów jak typografia, kolorystyka, odstępy i stany interakcji. Dzięki temu projekt nie wymagał definiowania wszystkich zasad od początku, a przygotowane widoki mogły opierać się na wspólnych regułach.

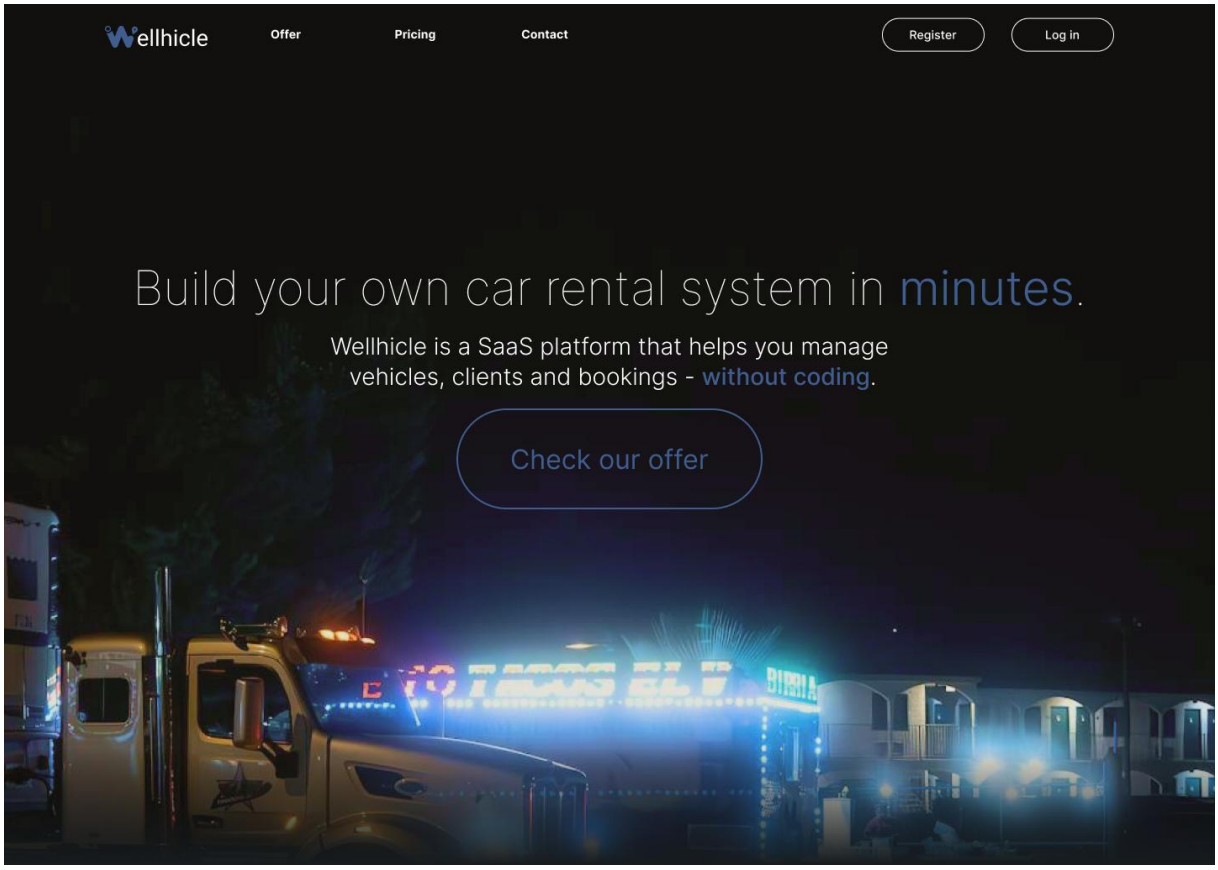
Material Theme Builder zastosowano do dopasowania motywu interfejsu do identyfikacji wizualnej systemu Wellhicle. Za jego pomocą dostosowano przede wszystkim kolorystykę oraz wybrane ustawienia typografii, zachowując jednocześnie strukturę systemu Material Design 3. Dzięki temu projektowane widoki miały spójny wygląd i mogły być łatwiej przenoszone do implementacji.

6.4.3.1. Widok strony startowej

Widok strony startowej zaprojektowano jako ekran wprowadzający do systemu Wellhicle. Jego zadaniem jest przedstawienie podstawowej idei platformy, wskazanie najważniejszych korzyści oraz skierowanie użytkownika do kolejnych działań takich jak sprawdzenie oferty, zapoznanie się z cennikiem subskrypcji, rejestracja lub logowanie.

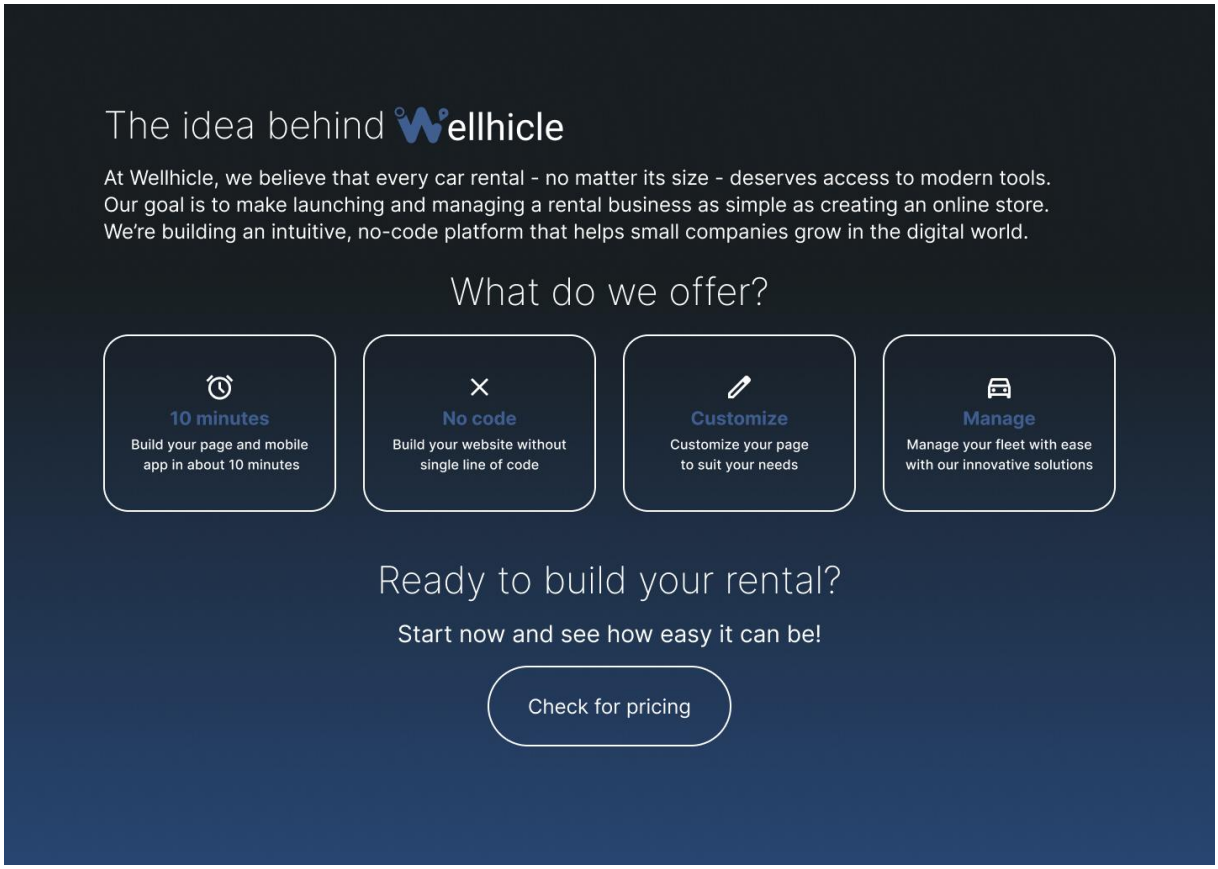
W górnej części strony przedstawionej na *Rysunek 82*, umieszczono pasek nawigacyjny z logo systemu, odnośnikami do głównych sekcji oraz przyciskami do logowania bądź rejestracji. Taki układ

pozwala użytkownikowi od razu przejść do interesujących go obszarów strony bez konieczności jej przewijania. Centralną część pierwszej sekcji zajmuje hasło opisujące główną funkcję platformy, a pod nim umieszczono krótki opis oraz przycisk prowadzący do przeglądu oferty.



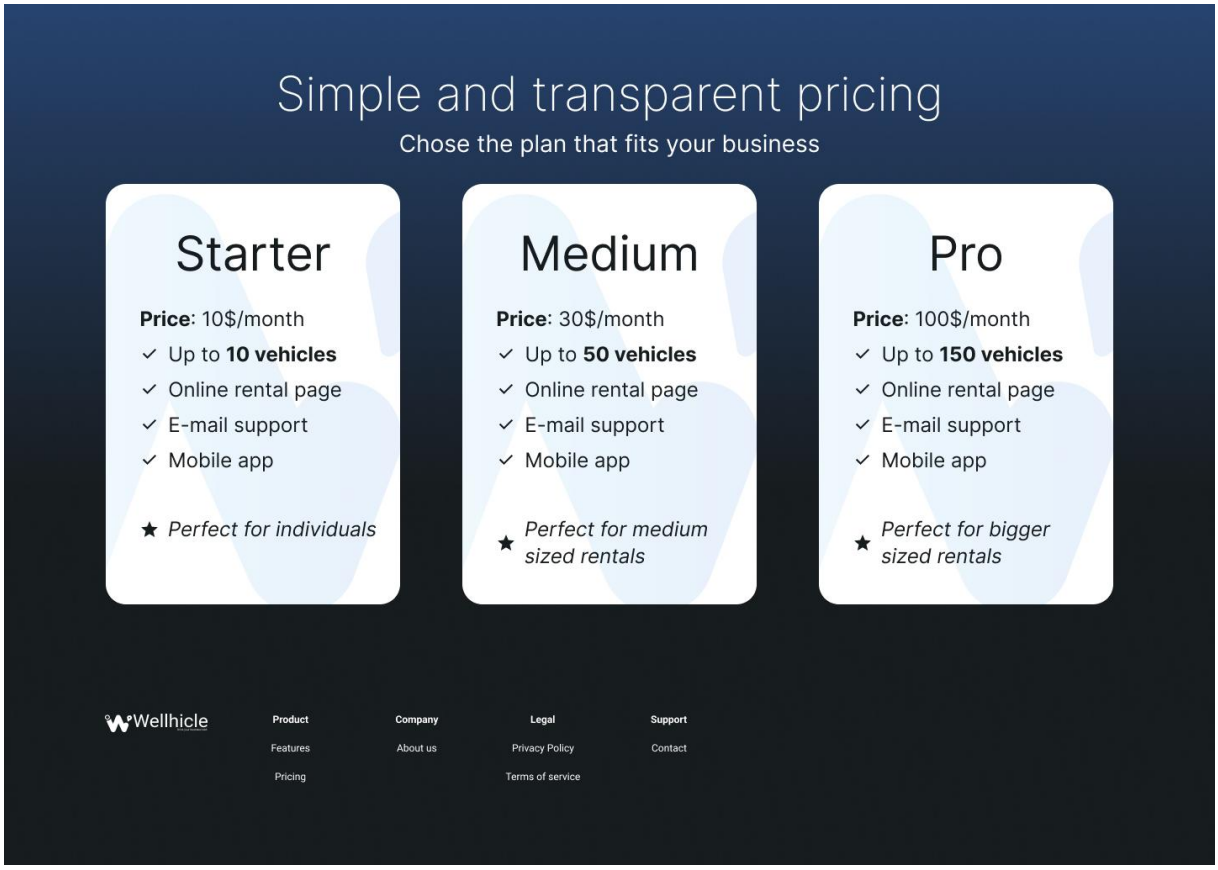
Rysunek 82 Projekt widoku górnej części strony startowej. Źródło: opracowanie własne.

Kolejna sekcja strony przedstawiona na *Rysunek 83,* wyjaśnia ogólne założenia systemu i jego przeznaczenie dla firm zajmujących się wynajmem pojazdów. W tej części zaprezentowano zestaw kart opisujących główne cechy platformy takie jak szybka konfiguracja, brak konieczności programowania, możliwość dostosowania strony internetowej wypożyczalni oraz zarządzanie flotą. Zastosowanie kart pozwala oddzielić najważniejsze informacje i ułatwia ich szybkie odczytanie. Pod kartami umieszczono krótki komunikat zachęcający do zapoznania się z ofertą serwisu, uzupełniony przyciskiem prowadzącym do sekcji pakietów.



Rysunek 83 Projekt widoku środkowej części strony startowej. Źródło: opracowanie własne.

Pakiety przedstawiono w formie kart cenowych, dzięki czemu użytkownik może porównać dostępne warianty pod względem zakresu oferowanych możliwości oraz przyjętych limitów. Widok strony startowej zamyka stopka zawierająca logo oraz odnośniki do podstawowych części serwisu, informacji, dokumentów prawnych i kontaktu. Widok został przedstawiony na *Rysunek 84.*



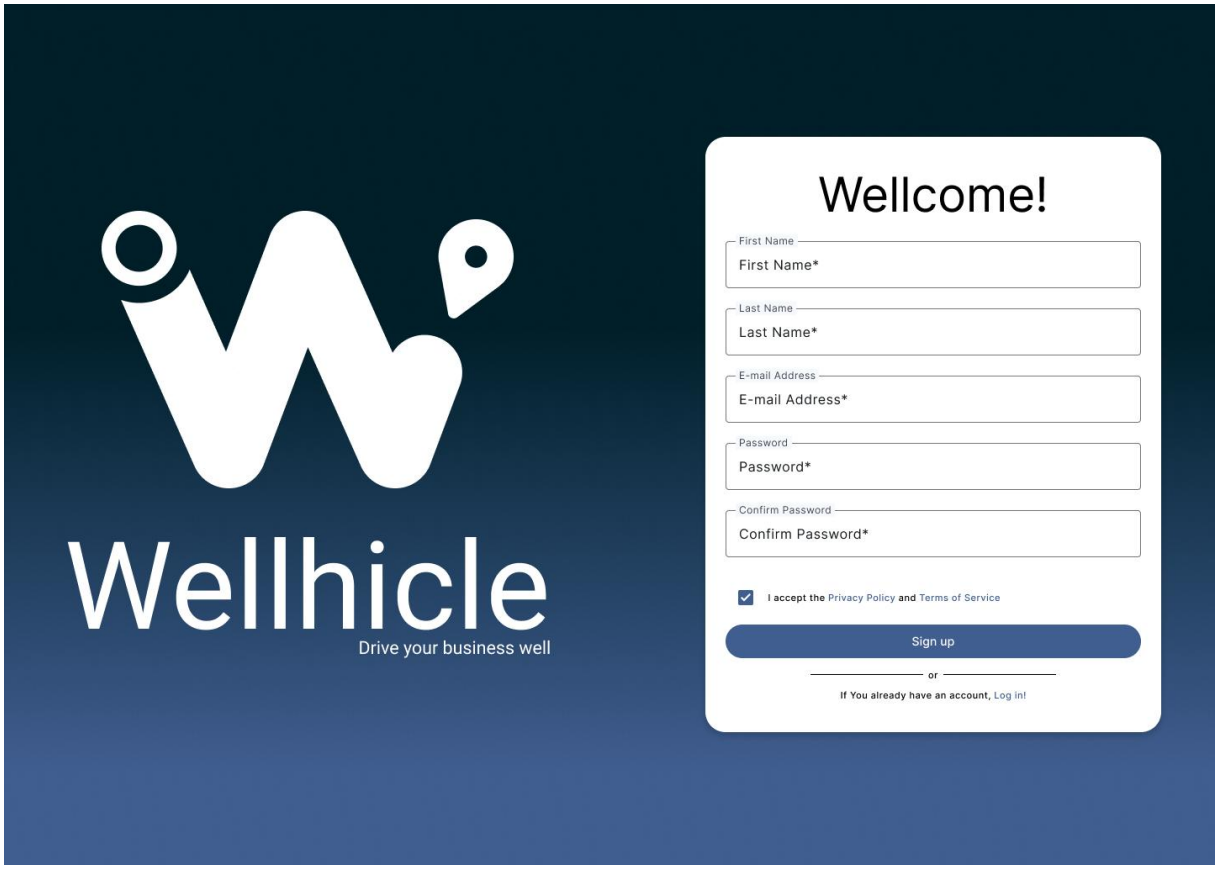
Rysunek 84 Projekt widoku końcowej części strony startowej. Źródło: opracowanie własne.

6.4.3.2. Widoki rejestracji i logowania

Widok rejestracji zaprezentowany na *Rysunek 85* zaprojektowano jako ekran umożliwiający tworzenie nowego konta użytkownika w systemie Wellhicle. Układ podzielono na dwie główne części. Po lewej stronie umieszczono element identyfikacji wizualnej systemu w postaci jednego z wariantów logo zaprezentowanych na *Rysunek 81*. Dzięki temu widok wskazuje, z jakim systemem użytkownik ma do czynienia.

Po prawej stronie umieszczono kartę formularza rejestracyjnego. Formularz obejmuje podstawowe dane wymagane do założenia konta, takie jak imię, nazwisko, adres e-mail, hasło oraz potwierdzenia hasła.

Pod polami formularza umieszczono pole wyboru służące do zaakceptowania polityki prywatności oraz regulaminu serwisu. Pod tą częścią znajduje się główny przycisk rejestracji. Na dole karty umieszczono również odnośnik prowadzący do logowania, przeznaczony dla użytkowników, którzy posiadają już konto w systemie.



Rysunek 85 Projekt widoku strony rejestracji. Źródło: opracowanie własne.

Widok logowania zaprezentowany na *Rysunek 86*, zachowuje układ zbliżony do widoku rejestracji w celu utrzymania spójności ekranów związanych z dostępem do systemu.

W porównaniu z widokiem rejestracji, formularz logowania ograniczono do dwóch pól: adresu e-mail oraz hasła. Pod nimi umieszczono główny przycisk logowania, a niżej odnośnik prowadzący do rejestracji nowego konta.

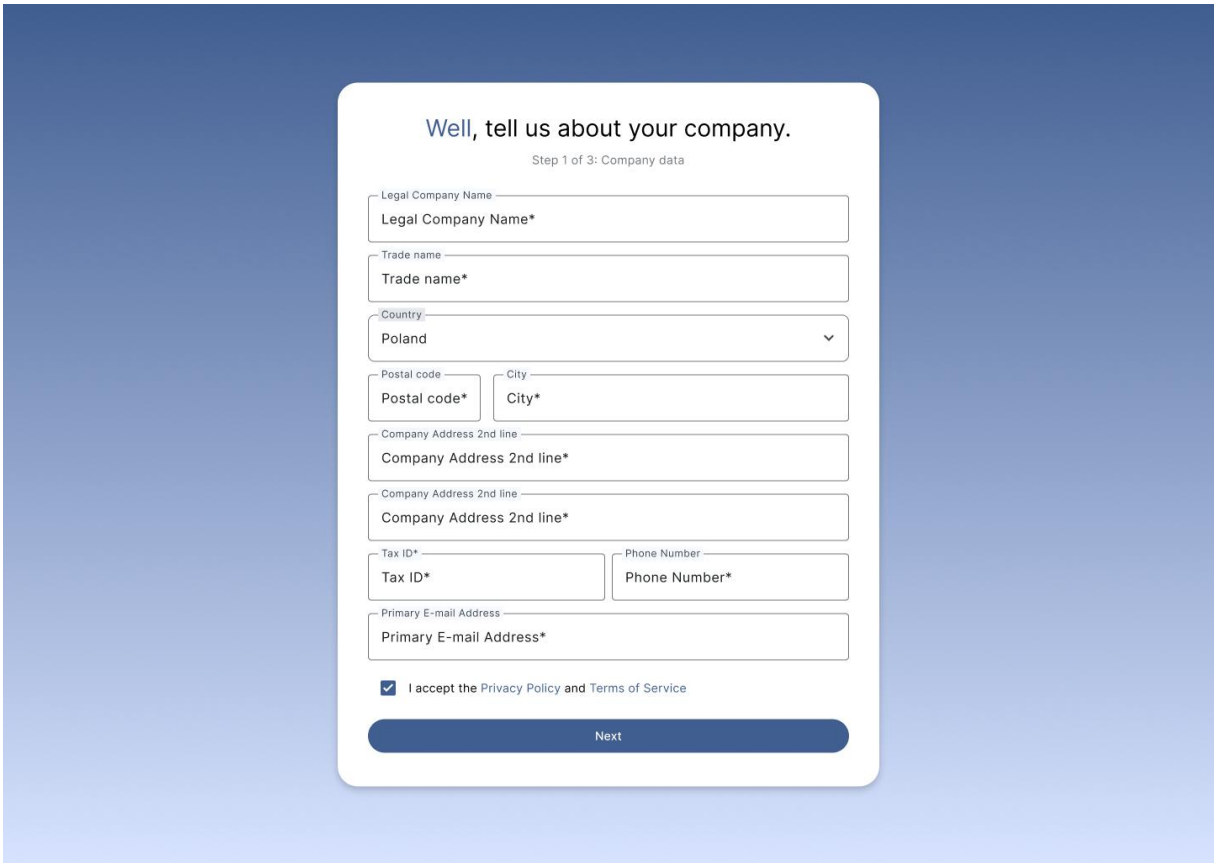


Rysunek 86 Projekt widoku strony logowania. Źródło: opracowanie własne.

6.4.3.3. Widok kreatora wypożyczalni

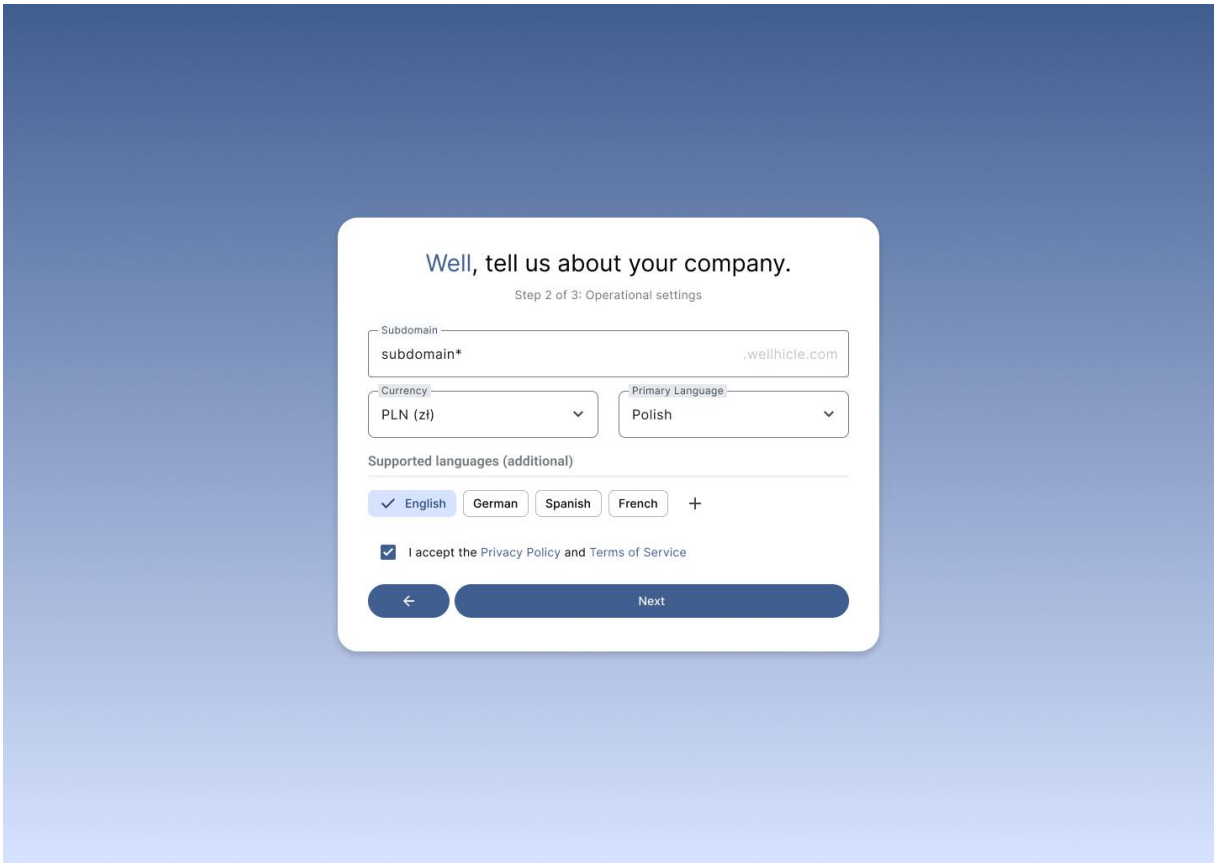
Widok kreatora wypożyczalni zaprojektowano jako formularz podzielony na trzy kolejne etapy. Jego celem jest zebranie danych do utworzenia i podstawowej konfiguracji nowej wypożyczalni w systemie Wellhicle. Formularz został podzielony na kroki w celu ograniczenia liczby informacji prezentowanych jednocześnie.

W pierwszym etapie przedstawionym na *Rysunek 87,* użytkownik uzupełnia podstawowe dane firmy. W dolnej części formularza znajduje się pole wyboru umożliwiające akceptację polityki prywatności oraz regulaminu serwisu, a zaraz pod nim jest przycisk umożliwiający przejście do następnego kroku formularza.



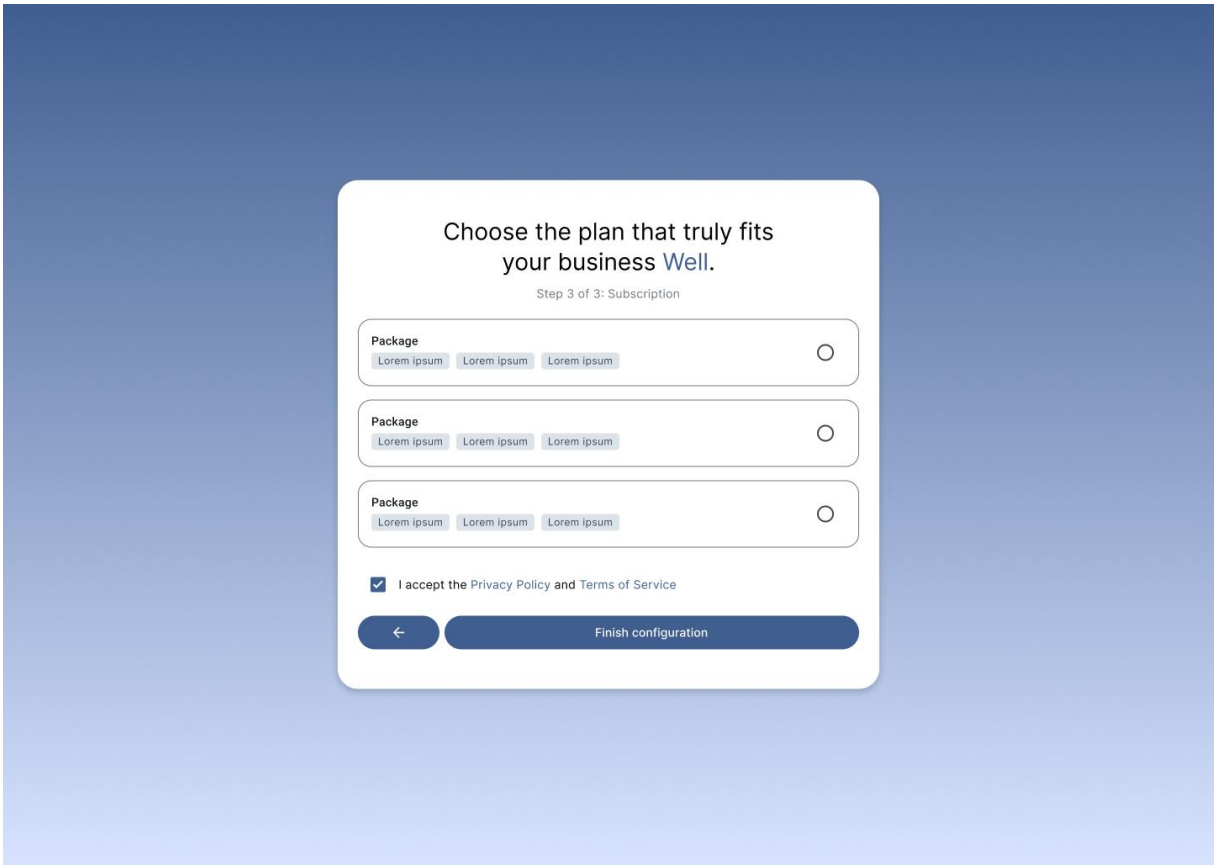
Rysunek 87 Projekt widoku pierwszego kroku kreatora wypożyczalni. Źródło: opracowanie własne.

Drugi etap przedstawiony na *Rysunek 88*, dotyczy ustawień wypożyczalni. Użytkownik podaje w nim subdomenę, pod którą zostanie udostępniona strona wypożyczalni, wybiera walutę, główny język systemu oraz dodatkowo obsługiwane języki. Waluta oraz główny język są wybierane z list rozwijanych, natomiast dodatkowo obsługiwane języki oznaczane są za pomocą przycisków typu chip. W dolnej części formularza zachowano ten sam układ akcji, dodając przycisk powrotu do poprzedniego kroku obok przycisku przejścia dalej. Dzięki temu użytkownik może poprawić wcześniej uzupełnione dane bez powtarzania całego procesu od nowa.



Rysunek 88 Projekt widoku drugiego kroku kreatora wypożyczalni. Źródło: opracowanie własne.

Trzeci a zarazem ostatni etap formularza zaprezentowany na *Rysunek 89*, służy do wyboru pakietu subskrypcji. Dostępne warianty zostały przedstawione jako oddzielne karty z krótkim opisem oraz możliwością wyboru jednego z nich. Na końcu formularza ponownie umieszczono pole wyboru akceptacji dokumentów, przycisk powrotu oraz główny przycisk zakończenia konfiguracji.



Rysunek 89 Projekt widoku trzeciego kroku kreatora wypożyczalni. Źródło: opracowanie własne.

Cały kreator zachowuje jednolity układ wizualny. Każdy etap jest prezentowany w centralnie umieszczonej karcie na tle w kolorach zgodnych z identyfikacją systemu. Nagłówek wskazuje aktualny krok, co ułatwia użytkownikowi orientację w procesie. Dzięki podziałowi formularza na trzy kroki jest on czytelniejszy, a użytkownik przechodzi przez konfigurację w uporządkowanej kolejności.

6.4.3.4. Widok listy

Widok listy został zaprojektowany na przykładzie listy pojazdów przedstawionej na *Rysunek 90*. Jego głównym zadaniem było umożliwienie przeglądania floty w celu odnalezienia konkretnego rekordu, który następnie mógł zostać otwarty w widoku szczegółów lub edytowany w widoku edycji.

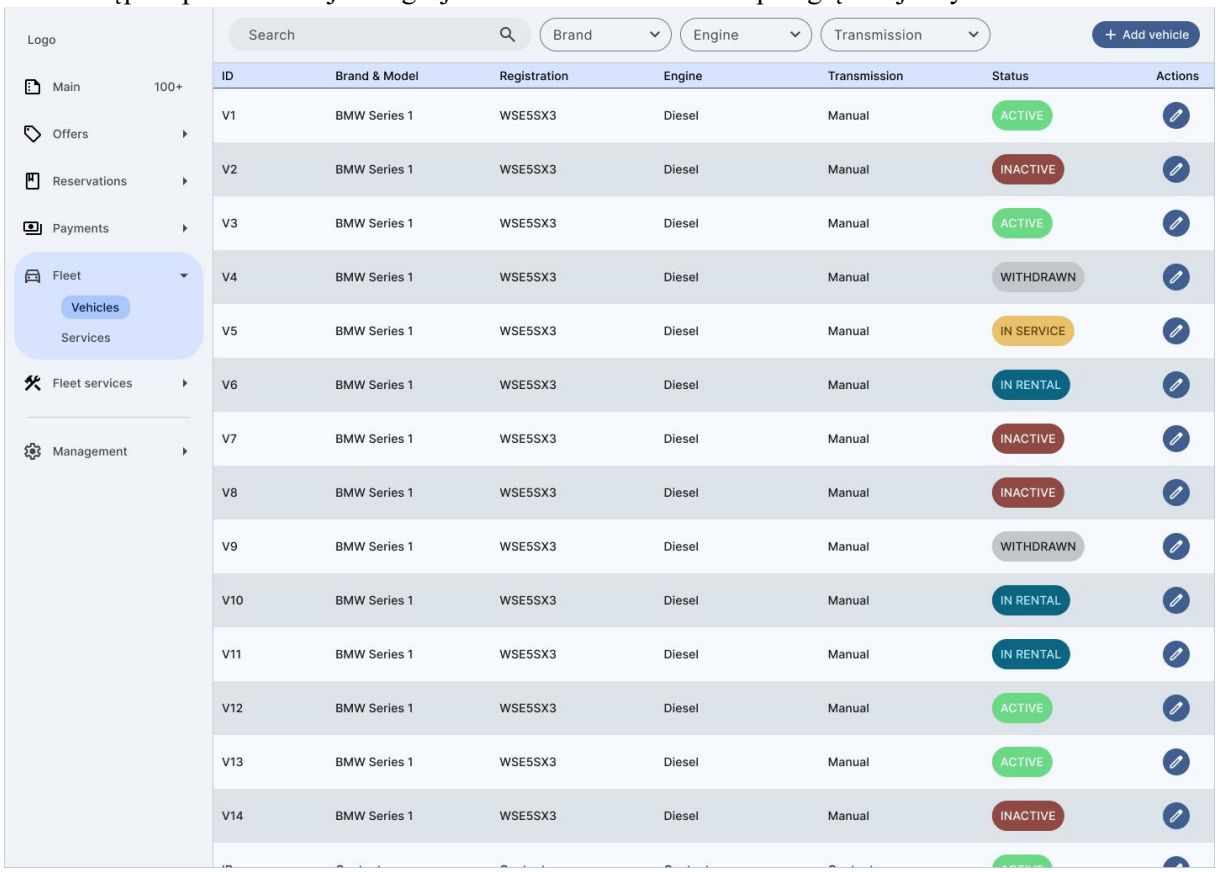
W górnej części widoku umieszczono pole wyszukiwania oraz filtry w formie list rozwijanych. Takie rozwiązanie ogranicza liczbę prezentowanych wyników, ułatwiając tym samym użytkownikowi dotarcie do szukanego rekordu. W prawym górnym rogu widoku znajduje się również przycisk dodania nowego pojazdu, ponieważ jest to jedna z podstawowych operacji wykonywanych z poziomu listy.

Dane pojazdów przedstawiono w formie tabeli, która zawiera najważniejsze informacje potrzebne do identyfikacji rekordu. Status przedstawiono w formie kolorowej etykiety tekstowej. Takie rozwiązanie zwiększa czytelność listy i pozwala szybciej rozróżnić poszczególne statusy pojazdu. Ostatnia kolumna została przeznaczona na akcję edycji dotyczącą konkretnego wiersza.

Ten widok pełni funkcje wzorca dla innych ekranów list w systemie Wellhicle. Powtarzalny układ obejmujący tabelę, filtry, statusy, wyszukiwarkę oraz edycję w wierszach ułatwia korzystanie z kolejnych modułów, ponieważ operacje prezentowane są w zbliżony sposób.

Stałym elementem widoków list jest również boczny panel nawigacyjny umieszczony po lewej stronie ekranu. Umożliwia on przechodzenie pomiędzy głównymi modułami systemu jednocześnie

wskazując użytkownikowi aktualne położenie w strukturze aplikacji. Dzięki temu użytkownik ma cały czas dostęp do podstawowej nawigacji niezależnie od aktualnie przeglądanej listy.



Rysunek 90 Projekt widoku listy pojazdów. Źródło: opracowanie własne.

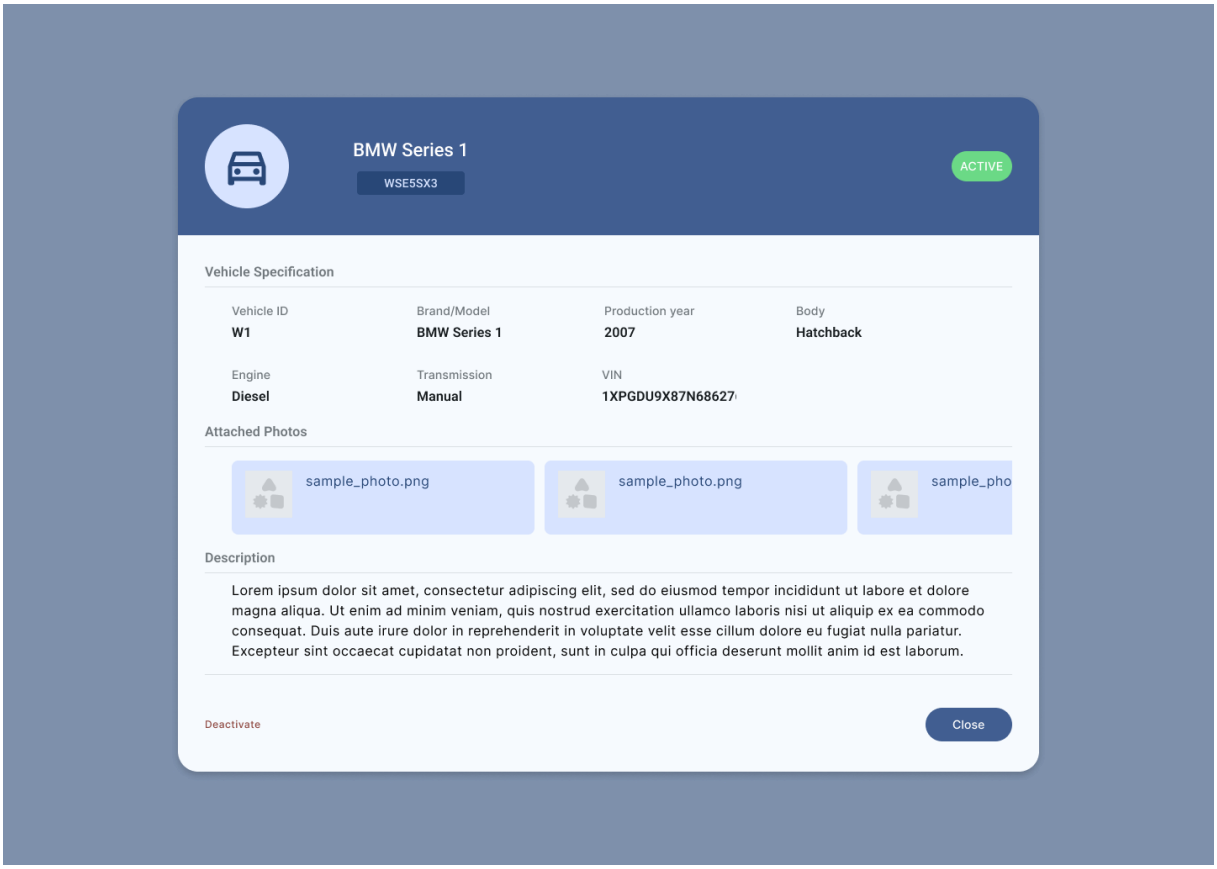
6.4.3.5. Widok szczegółów

Widok szczegółów zaprezentowany na *Rysunek 91*, zaprojektowano jako okno dialogowe służące do prezentacji wszystkich danych wybranego pojazdu. Zastosowanie okna dialogowego w przypadku tego widoku pozwala użytkownikowi skupić uwagę na wybranym pojeździe bez całkowitego opuszczania kontekstu listy.

W nagłówku widoku umieszczone zostały podstawowe informacje identyfikujące pojazd. Dzięki temu użytkownik od razu widzi, którego rekordu dotyczą prezentowane dane. Pozostałe informacje podzielono na sekcje, obejmujące specyfikację pojazdu, załączone zdjęcia oraz opis.

W dolnej części widoku przewidziano akcję zamknięcia okna oraz osobną akcję zmiany stanu pojazdu. Ustawienie tych operacji w dużej odległości od siebie jest istotne, ponieważ akcja zamknięcia widoku ma charakter neutralny z punktu widzenia systemu, natomiast zmiana stanu rekordu może wpłynąć na dalsze jego działanie.

Widok szczegółów stanowi wzorzec dla innych ekranów służących do prezentacji danych pojedynczego rekordu. Powtarzalny układ z nagłówkiem, sekcjami informacyjnymi oraz przyciskami akcji ułatwia korzystanie z systemu, ponieważ użytkownik otrzymuje szczegóły obiektów w podobnej strukturze niezależnie od modułu.



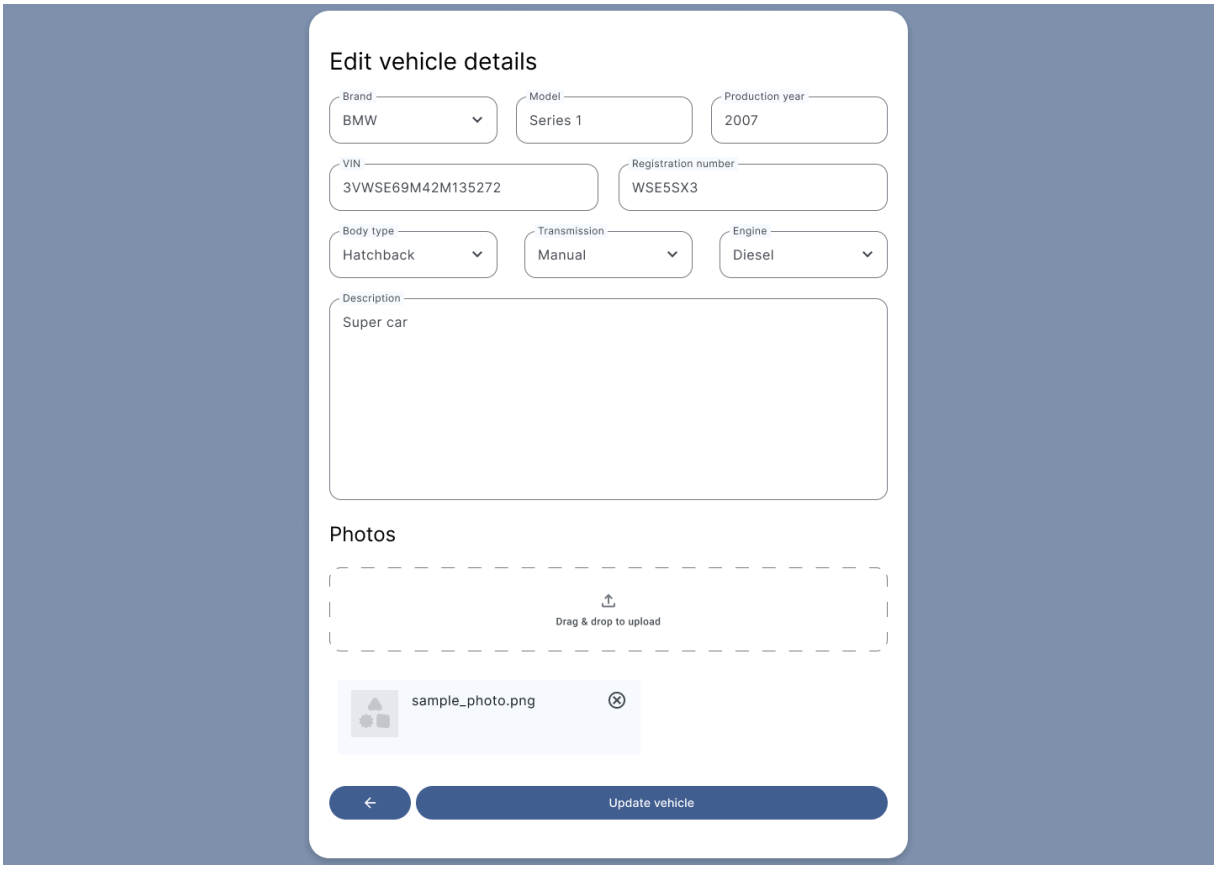
Rysunek 91Projekt widoku szczegółów pojazdu. Źródło: opracowanie własne.

6.4.3.6. Widok edycji

Widok edycji przedstawiony na *Rysunek 92*, przygotowano jako formularz umożliwiający modyfikację danych pojazdu. Zastosowano kartę dialogową z wieloma polami formularza, które umożliwiają użytkownikowi zmianę danych istniejącego już pojazdu.

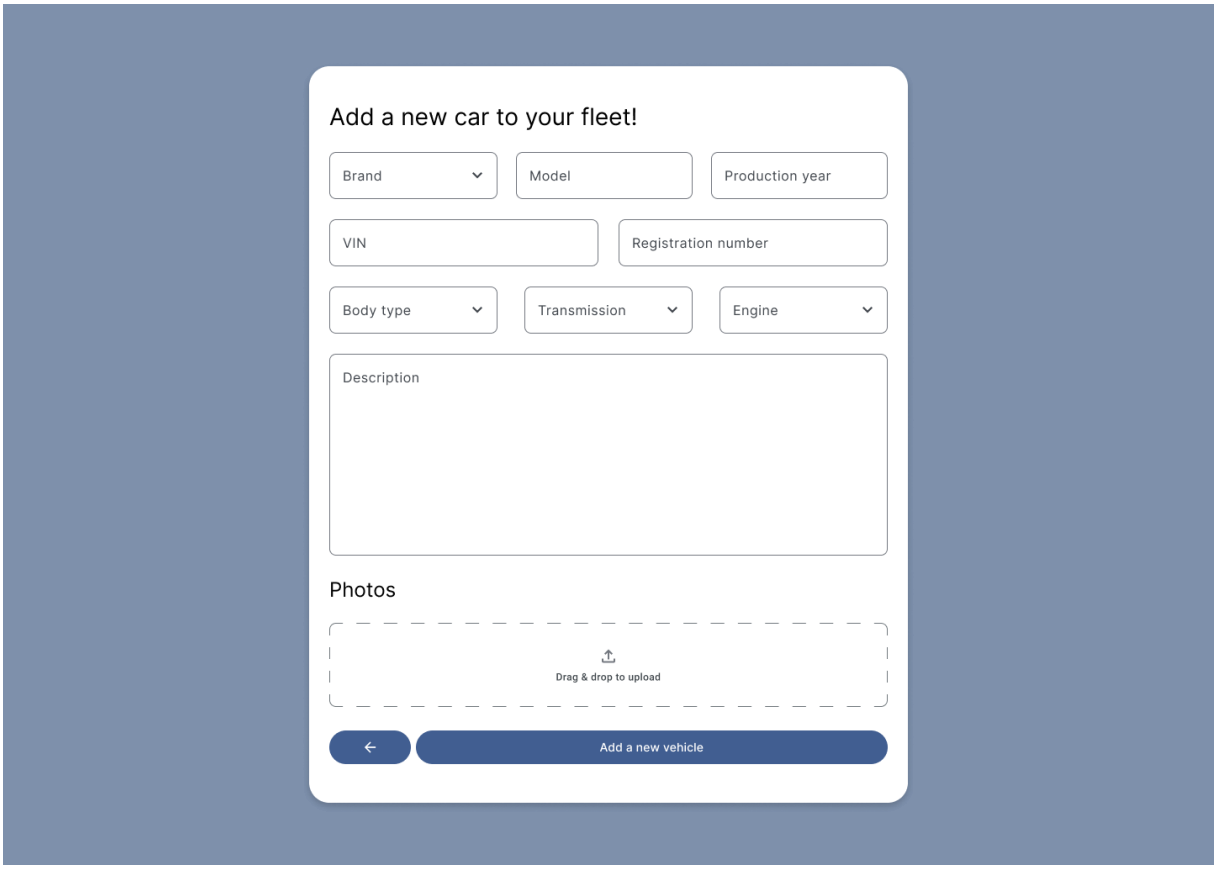
W formularzu wykorzystano różne typy pól, dopasowane do charakteru wprowadzanych danych. Dla wartości wybieranych z ograniczonego zbioru zastosowano listy rozwijane, natomiast dla danych indywidualnych, takich jak model, numer VIN czy numer rejestracyjny, przewidziano pola tekstowe. Takie rozróżnienie zmniejsza ryzyko wprowadzania niespójnych danych i porządkuje sposób uzupełniania formularza.

Osobną część formularza przeznaczono na zdjęcia pojazdu. Zaprojektowano obszar dodawania plików oraz listę już załączonych elementów z możliwością ich usunięcia przed zapisaniem zmian. Formularz w dolnej części udostępnia także akcję powrotu oraz główny przycisk zapisania zmian, co pozwala oddzielić rezygnację z edycji od zatwierdzenia formularza.



Rysunek 92 Projekt widoku edycji pojazdu. Źródło: opracowanie własne.

Bliźniaczym widokiem do edycji pojazdu jest widok dodania nowego pojazdu przedstawiony na *Rysunek 93*. Oba widoki wykorzystują ten sam układ formularza oraz obejmują ten sam zakres pól. Różnice pomiędzy tymi dwoma widokami wynikają wyłącznie z kontekstu użycia. W formularzu dodawania pola nie są wcześniej uzupełnione, a tytuł okna oraz przycisk głównej akcji dotyczą dodania nowego pojazdu.



Rysunek 93 Projekt widoku dodania pojazdu. Źródło: opracowanie własne.

6.4.4. Zakres i rola przygotowanego projektu interfejsu

W założeniach przyjęto, że projekt interfejsu powinien docelowo obejmować całość systemu, tak aby poszczególne jego obszary były tworzone według wspólnych zasad wizualnych i funkcjonalnych. W praktyce zakres prac projektu został ograniczony do wybranych ekranów, które pozwoliły określić główny kierunek wizualny systemu oraz sposób organizacji najważniejszych elementów interfejsu.

Opracowane projekty widoków pozwoliły ustalić podstawowy układ ekranów, sposób prezentacji danych, charakter formularzy, rozmieszczenie akcji oraz ogólne zasady wizualne stosowane w systemie. Nie przygotowano jednak kompletnego zestawu widoków dla wszystkich modułów aplikacji, dlatego część rozwiązań musiała być doprecyzowana dopiero na etapie implementacji.

Prace nad projektem interfejsu zakończono na wcześniejszym etapie niż było to pierwotnie planowane, ponieważ dalszy rozwój systemu wymagał zaangażowania zespołu w implementację oraz integracje poszczególnych części aplikacji. Z tego względu przygotowany projekt należy traktować jako podstawę kierunkową dla dalszych prac nad interfejsem, a nie jako kompletną dokumentację projektową całości aplikacji. Mimo ograniczeń, projekty widoków spełniły swoją najważniejszą funkcję, ponieważ wyznaczyły ogólny kierunek wizualny systemu. Szczegółowe rozwiązania dla części modułów były następnie rozwijane bezpośrednio na etapie implementacji.

7. Implementacja

Implementacja systemu Wellhicle obejmuje pięć głównych komponentów: serwer aplikacyjny, aplikacje przeglądarkowe, aplikację mobilną, infrastrukturę wdrożeniową oraz integrację z zewnętrznym systemem ERP.

Serwer aplikacyjny, opisany w podrozdziale *7.1*, odpowiada za logikę domenową, reguły biznesowe, autoryzację oraz dostęp do danych. Zastosowano w nim architekturę warstwową oraz wzorzec CQRS, co pozwoliło rozdzielić odpowiedzialności poszczególnych części systemu.

Aplikacje przeglądarkowe, przedstawione w podrozdziale *7.2*, obejmują interfejsy przeznaczone dla różnych grup użytkowników: pracowników wypożyczalni, administratorów platformy, klientów wypożyczalni oraz osób zapoznających się z ofertą platformy. Część elementów wspólnych wydzielono do osobnej biblioteki, co ułatwiło ponowne wykorzystanie komponentów i zachowanie spójności między aplikacjami.

Aplikacja mobilna, omówiona w podrozdziale *7.3*, wspiera procesy operacyjne realizowane poza środowiskiem przeglądarkowym, w szczególności zadania pracowników operacyjnych. Komunikuje się ona z serwerem aplikacyjnym za pośrednictwem interfejsu REST API.

Infrastruktura wdrożeniowa, opisana w podrozdziale *7.4*, została przygotowana z wykorzystaniem konteneryzacji, automatyzacji procesu budowania i wdrażania oraz narzędzi do monitorowania działania systemu. Dzięki temu możliwe jest uporządkowane uruchamianie, aktualizowanie i obserwowanie poszczególnych komponentów systemu.

7.1. Serwer aplikacyjny

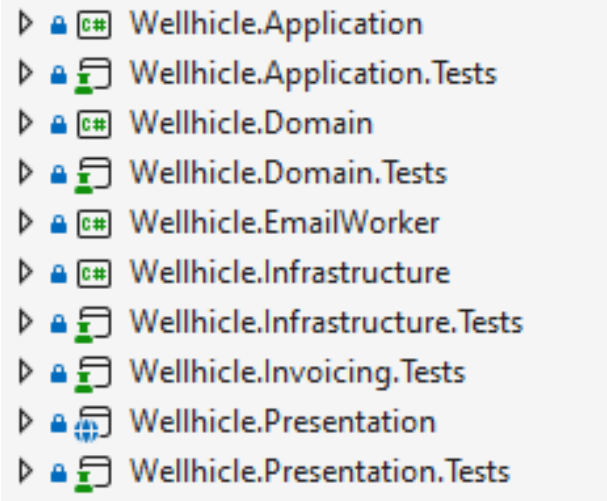
Serwer aplikacyjny Wellhicle stanowi centralny komponent platformy. Realizuje całość logiki biznesowej i pośredniczy między aplikacjami klienckimi – webowym panelem tenanta, aplikacją dla klienta wypożyczalni, panelem administracyjnym, mobilną aplikacją operacyjną oraz klientem systemu ERP enova365 – a wspólną relacyjną bazą danych. Komunikacja odbywa się przez interfejs REST API zgodny ze specyfikacją OpenAPI, uwierzytelnianie opiera się na tokenach JWT, a izolacja danych pomiędzy klientami egzekwowana jest na każdym etapie przetwarzania zgodnie z modelem wielotenantowym (SaaS). Architektura aplikacji opiera się na zasadach czystej architektury z uproszczonym wariantem wzorca CQRS, a operacje wykraczające poza cykl żądanie-odpowiedź zostały wydzielone do osobnego workera i procesów harmonogramowanych.

7.1.1. Struktura rozwiązania w .NET

Rozwiązanie aplikacji serwerowej zorganizowano zgodnie z modelem czystej architektury (ang. *Clean Architecture*), w którym logika biznesowa stanowi rdzeń systemu. Otoczona jest warstwami technicznymi. Warstwy te są od niej zależne i nie posiadają bezpośrednich zależności do mechanizmów wewnętrznych. Dostęp realizowany jest wyłącznie przez interfejsy definiowane bliżej centrum systemu [101].

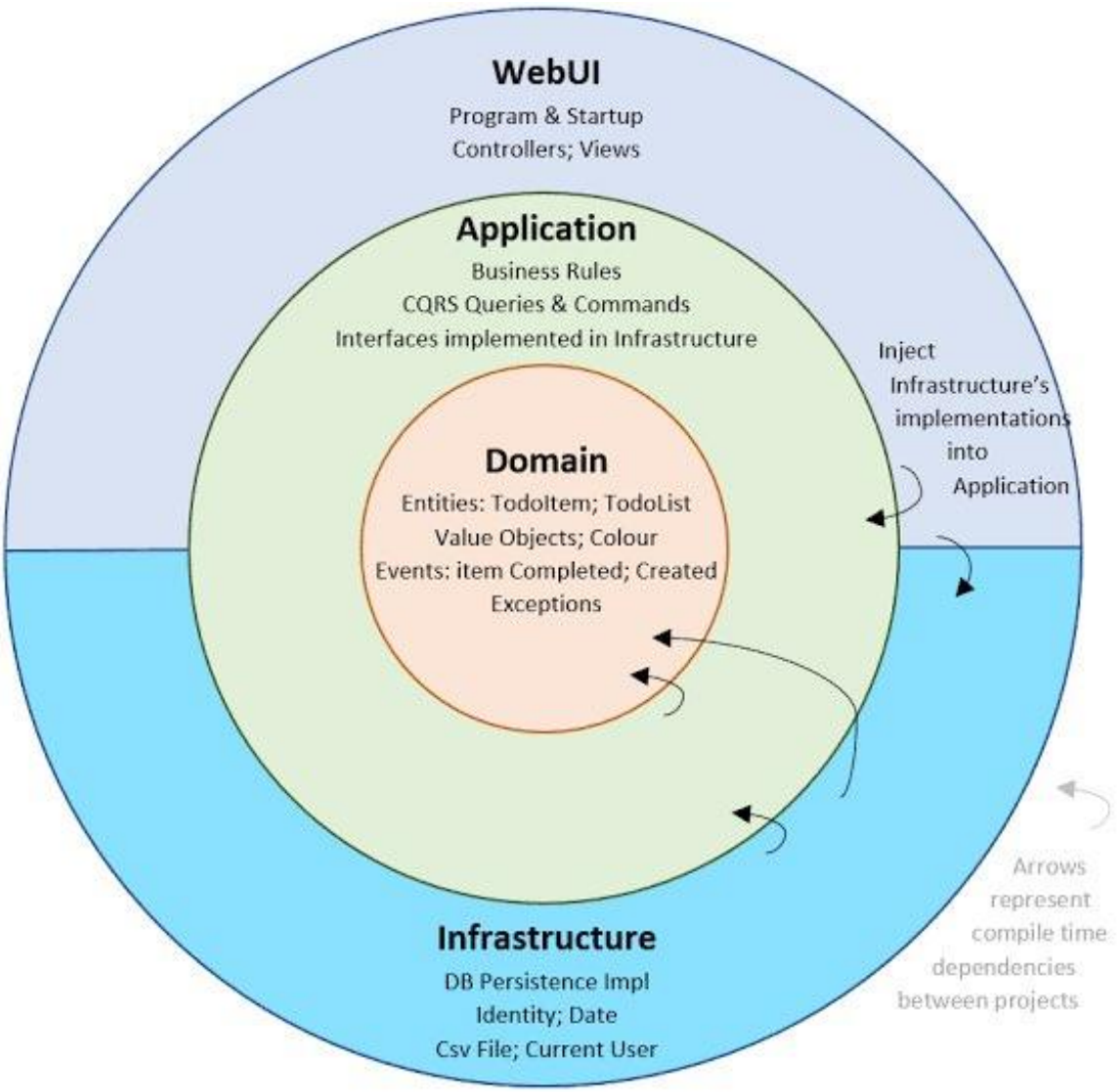
W architekturze wyróżniono pięć odrębnych projektów .NET (*Rysunek 94*). Odpowiadają kolejnym warstwom: `Wellhicle.Domain` zawiera model biznesowy (encje, agregaty i zdarzenia domenowe), `Wellhicle.Application` realizuje przypadki użycia w uproszczonym CQRS z wykorzystaniem kontraktów opisanych w punkcie *7.1.3*, `Wellhicle.Infrastructure` implementuje te kontrakty przy użyciu Entity Framework Core, brokera wiadomości (RabbitMQ) i magazynu plików zgodnego z protokołem S3, `Wellhicle.Presentation` wystawia API HTTP

za pomocą ASP.NET Core, a `Wellhicle.EmailWorker` obsługuje asynchroniczne zadania poza głównym cyklem żądań i odpowiedzi (np. przetwarzanie wiadomości e-mail). Każdy projekt wyposażono również we własny zestaw testów jednostkowych. Gwarantuje to izolację poszczególnych komponentów systemu [102].



Rysunek 94 Struktura rozwiązania w Visual Studio. Źródło: opracowanie własne.

Kierunek zależności jest egzekwowany na poziomie plików projektu. Projekt `Wellhicle.Domain` nie zawiera żadnych referencji do innych projektów rozwiązania. Oznacza to, że rdzeń modelu biznesowego pozostaje abstrakcyjny wobec pojęć takich jak kontroler, repozytorium czy transakcje bazy danych. `Wellhicle.Application` odwołuje się wyłącznie do `Wellhicle.Domain` oraz definiuje kluczowe interfejsy implementowane przez warstwę infrastruktury – `Wellhicle.Infrastructure`. Warstwa prezentacji oraz serwis roboczy zależą od warstwy aplikacyjnej i pośrednio od infrastruktury, korzystając wyłącznie z ich kontraktów, podczas gdy konkretne implementacje są dostarczane przez mechanizm wstrzykiwania zależności. Kontrolery w warstwie prezentacji pełnią rolę cienkich adapterów. Wstrzykują interfejsy `ICommandHandler` i `IQueryHandler` przez konstruktor i przekazują do nich żądania bez własnej logiki biznesowej. *Rysunek 95* przedstawia wizualizację zależności pomiędzy poszczególnymi warstwami aplikacji.



Rysunek 95 Struktura warstw modelu Clean Architecture. Źródło: [103].

Należy odróżnić kierunek kompilacyjnych zależności od kolejności wywołań podczas działania systemu. Żądanie HTTP inicjuje przepływ danych przez warstwy: prezentacji, handlera w warstwie aplikacyjnej, operacje na agregatach domeny oraz interfejsy sięgające do infrastruktury (np. zapis danych). Jednak zależności kodowe wskazują odwrotnie: prezentacja zna warstwę aplikacyjną, ta z kolei domenową. Sama infrastruktura zaś implementuje logikę biznesową. Taka architektura umożliwia łatwe wymiany technologii (np. zmiana bazy danych na inną niż SQL Server) bez ingerencji w model domenowy czy przypadki użycia. Wystarczy podmienić implementacje, zachowując abstrakcyjność warstwy biznesowej.

7.1.2. Warstwa domeny

Domena projektu Wellhicle obejmuje kilkanaście powiązanych ze sobą obszarów biznesowych od użytkowników i tenantów, przez pojazdy, oferty i rezerwacje, po rozliczenia i subskrypcje. Każdy z nich ma własne reguły, niezmienniki i cykl życia obiektów.

7.1.2.1. Rola warstwy domeny w projekcie

Warstwa domeny w aplikacji serwerowej Wellhicle skupia kod opisujący model biznesowy platformy: pojęcia, ich relacje oraz reguły zmiany stanu. Stanowi rdzeń architektury aplikacji, gdzie żaden element nie zależy od konkretnej technologii, a wszystkie pozostałe warstwy odwołują się do domeny. Taka organizacja zgodna jest z regułą zależności z *Clean Architecture* [104].

Niezależność technologiczna jest egzekwowana już na poziomie pliku projektu. Plik konfiguracyjny *Wellhicle.Domain.csproj* nie zawiera żadnych referencji do innych projektów rozwiązania ani do paczek związanych z ASP.NET Core, Entity Framework Core czy klientami HTTP.

Domena jest zorganizowana wokół obszarów biznesowych odpowiadających następującym modułom: Tenants, Users, Vehicles, VehicleGroups, VehicleHandovers, Reservations, Offers, Billings, Payments, Subscriptions, Files, Emails, Departments, ReferenceData oraz wspólny katalog Common z elementami bazowymi. Taka struktura umożliwia rekonstrukcję działania systemu na podstawie analizy warstwy domeny, bez konieczności sięgania do kontrolerów czy infrastruktury.

7.1.2.2. Elementy bazowe modelu domenowego

W przestrzeni nazw `Wellhicle.Domain.Common` zdefiniowano klasy bazowe, które stanowią podstawę dla obiektów domenowych wszystkich modułów: `Entity`, `AggregateRoot`, `ValueObject`, `DomainEvent` oraz dwa interfejsy znacznikowe `IAuditable` i `ITenantOwned`. Terminologia ta odzwierciedla klasyczne pojęcia z modelowania zorientowanego na domenę (ang. *Domain-Driven Design*) [105]. Encja posiada własną tożsamość niezależną od atrybutów. Korzeń agregatu wyznacza granicę spójności i jest jedynym punktem modyfikacji stanu całego agregatu. Obiekt wartości identyfikowany jest przez zestaw składowych. Zdarzenie domenowe opisuje fakt zaistniały w modelu.

Klasa Entity wprowadza tożsamość poprzez właściwość `Id`, inicjalizowaną podczas zapisu do bazy, oraz przesłonięcia metod `Equals`, `GetHashCode` i operatorów porównania „==” oraz „*!=*”, opartych na identyfikatorze. Encja niezapisana (z atrybutem Id równym 0) traktowana jest jako obiekt przejściowy. Dwa takie obiekty, nawet o identycznych wartościach atrybutów, nie są uznawane za równe. Rozróżnienie to ma znaczenie praktyczne. Pozwala odróżnić sytuację „dwa identyczne wpisy w formularzu” od sytuacji „dwa różne odwołania do tego samego rekordu w bazie”.

Klasa `AggregateRoot` (*Listing 2*), rozszerzająca `Entity`, zawiera cztery elementy o znaczeniu architektonicznym. `TenantId` stanowi część tożsamości biznesowej tych agregatów, które należą do konkretnego klienta [106]. Jest to sytuacja domyślna w modelu SaaS, ale nie obejmuje agregatów platformowych takich jak `Admin`, działających ponad tenantami. `RowVersion` to znacznik wersji wykorzystywany przez Entity Framework do optymistycznej kontroli współbieżności. Kolekcję zdarzeń domenowych z chronioną metodą `AddDomainEvent` służącą do ich rejestrowania oraz publiczną metodę `ClearDomainEvents`, używaną przez warstwę infrastruktury po publikacji. Identyfikator klienta nie jest dopisywany dopiero na etapie utrwalania danych, lecz stanowi integralną część definicji agregatu.

Listing 2 Klasa `AggregateRoot` z projektu `Wellhicle.Domain`. Źródło: opracowanie własne.

```
namespace Wellhicle.Domain.Common;

public abstract class AggregateRoot : Entity, ITenantOwned
{
   public int TenantId { get; protected set; }

    [Timestamp]
    public byte[] RowVersion { get; private set; } = Array.Empty<byte>();

    private readonly List<IDomainEvent> _domainEvents = new();

    [NotMapped]
    public Is.AsReadOnly();

    protected AggregateRoot(int tenantId)
    {
        if (tenantId < 0) throw new ArgumentException("TenantId cannot be
negative.", nameof(tenantId));

        TenantId = tenantId;
    }

    protected AggregateRoot() { }

    protected void AddDomainEvent(IDomainEvent @event)
    {
        if (@event is null) throw new
ArgumentNullException(nameof(@event));
        _domainEvents.Add(@event);
    }

    public void ClearDomainEvents() => _domainEvents.Clear();
}
```

`ValueObject` definiuje porównywanie obiektów na podstawie zestawu komponentów zwracanych przez chronioną metodę `GetEqualityComponents()`. Klasa ta stanowi fundament dla typów wartości, które są wprowadzane jako zamienniki typów prymitywnych. Najprostszym konkretnym przykładem jest klasa `Email` przedstawiona na *Listing 3*. Zawiera ona walidację składni (sprawdzenie pojedynczego znaku „@” w niezerowej pozycji środkowej) oraz normalizację (usunięcie białych znaków i konwersja na małe litery). Dzięki temu obiekt domenowy operuje na uporządkowanej wartości, a nie surowym ciągu znaków.

Listing 3 Klasa `Email` z projektu `Wellhicle.Domain`. Źródło: opracowanie własne.

```
namespace Wellhicle.Domain.Common;

public class Email : ValueObject
{
    public string Value { get; private set; }

    public Email(string value)
    {
        if (string.IsNullOrWhiteSpace(value))
            throw new ArgumentException("Email cannot be empty.",
nameof(value));

        var normalizedValue = value.Trim().ToLowerInvariant();
        if (!IsValidEmail(normalizedValue))
            throw new ArgumentException("Invalid email format.",
nameof(value));
```

```
        Value = normalizedValue;
    }

    private static bool IsValidEmail(string email)
    {
        if (string.IsNullOrWhiteSpace(email)) return false;

        int index = email.IndexOf('@');

        return index > 0 && index != email.Length - 1 && index ==
email.LastIndexOf('@');
    }

    protected override IEnumerable<object> GetEqualityComponents() {
yield return Value; }
    public override string ToString() => Value;
}
```

Analogiczną rolę pełnią *PhoneNumber*, *TimeZoneId*, *Address* oraz inne typy wartości specyficzne dla obszarów (np. *PersonalDocument*, *CompanyData*). Każdy z nich nie może przyjmować stanu niepoprawnego. Jeśli konstruktor odrzuci wartość, obiekt nigdy nie powstaje. Reguły walidacji są tym samym zapisane raz, bezpośrednio u źródła pojęcia, a nie rozproszone w handlerach.

DomainEvent to niemodyfikowalny rekord zawierający pola takie jak *TimeOccurred*, *EventType*, *AggregateId*, *TenantId*, *CorrelationId*, *CausationId* oraz *Version*. Agregaty dodają zdarzenia za pomocą chronionej metody *AddDomainEvent*, a warstwa aplikacji publikuje je po pomyślnym zapisie. Model uzupełniają dwa interfejsy znacznikowe. *IAuditable* deklaruje pola *CreatedAt* i *UpdatedAt*, ale nie narzuca obowiązku ich wypełniania (są one inicjalizowane przez warstwę utrwalania danych). *ITenantOwned* jest interfejsem pustym, wykorzystywanym przez infrastrukturę do automatycznego ustawiania *TenantId* na nowych encjach.

Model dopuszcza również obiekty o zasięgu globalnym. Przykładowe klasy, takie jak *EmailTemplate*, *Translation* czy *ProductPrice*, mają pole *TenantId* zadeklarowane jako nullowalny „*int?*”. Wartość *null* oznacza rekord systemowy wspólny dla całej platformy (np. domyślny szablon e-maila lub referencyjny cennik), natomiast wartość ustawiona reprezentuje rekord prywatny przypisany do konkretnego klienta. Klient może nadpisywać rekordy globalne własnymi wersjami. Mechanizm rozróżniania zasięgu jest integralną częścią semantyki tych klas, a nie warstwy zapytań.

7.1.2.3. Główne agregaty domeny

Agregat *Tenant* stanowi organizacyjny rdzeń modelu i reprezentuje klienta korzystającego z platformy SaaS. Przechowuje subdomenę, opcjonalną domenę własną oraz flagę informującą o statusie weryfikacji tej domeny, dane firmy, obsługiwane języki i waluty (z wyraźnym wskazaniem podstawowej waluty i języka), nazwę handlową, ustawienia witryny oraz status aktywności. Agregat ten pełni funkcję punktu odniesienia dla większości pozostałych elementów modelu. Jego relacje zostały opisane szczegółowo w podpunkcie *6.3.3.1*.

Specyfika inicjalizacji *TenantId* wynika ze struktury modelu. Konstruktor wywołuje metodę *base(tenantId: 0)*, a po wygenerowaniu identyfikatora przez bazę danych wartość ta jest wiązana z *Id* poprzez metodę *BindTenantIdToId()*. Rozwiązanie to zostało narzucone koniecznością spełnienia warunku, że TenantId agregatu musi być równy jego własnemu *Id*, co nie jest

możliwe do osiągnięcia w konstruktorze ze względu na fakt, iż `Id` jest przypisywane dopiero podczas zapisu obiektu.

`User` jest abstrakcyjną klasą bazową konta użytkownika. Definiuje wspólne pola oraz operacje dla różnych typów kont. Są trzy typy dziedziczące po `User`. `Client` (klient wypożyczalni, z opcjonalnym językiem, adresem, numerem telefonu, dokumentem tożsamości i danymi firmy oraz osobnymi flagami `IsActive` dla konta i `IsProfileActive` dla profilu, co pozwala zachować historyczne dane klienta przy dezaktywacji jego dostępu), `Employee` (pracownik organizacji powiązany z działem, z numerem telefonu i podpisem dokumentowym) oraz `Admin` (administrator platformy działający poza kontekstem pojedynczego tenanta).

`Reservation` modeluje rezerwację z rozbudowanym cyklem życia. Przechowuje dwie pary dat planowanych. Jedna reprezentuje deklarację w momencie utworzenia, a druga jest aktualizowana przy każdej zmianie harmonogramu. Różnica między nimi umożliwia śledzenie modyfikacji podczas obsługi rezerwacji. Status (`PLANNED`, `PENDING`, `CONFIRMED`, `ACTIVE`, `COMPLETED`, `CANCELED`, `NO_SHOW`, `INTERRUPTED`) zmienia się w ściśle określony sposób. Fazy wydania pojazdu i zwrotu są modelowane jako odrębne etapy. Każdy ma własne statusy oraz daty. Powiązanie z fakturą jest realizowane w relacji jeden do jednego poprzez `BillingItem`. Opcjonalne użycie pojazdu w rezerwacji reprezentuje klasa `VehicleUsage` (rezerwacja anulowana może nie mieć przypisanego użycia). To rozróżnienie jest uwzględnione w samym agregacie.

`Vehicle` reprezentuje pojazd floty. Zawiera metadane techniczne takie jak VIN, rok produkcji, markę i model (określony przez referencję do `VehicleModel`), typ nadwozia oraz rodzaj paliwa. Klasa obejmuje również powiązania operacyjne: kolekcję `VehicleDepartment` (przypisującą pojazd do działów z określonymi przedziałami czasowymi), `VehicleUsage` (rejestrującą poszczególne wykorzystania, np. rezerwacje czy serwisy) oraz `VehicleGroup` (służącą grupowaniu dla celów cennikowych i raportowania). Dodatkowo zawiera `VehicleCondition`, umożliwiając generowanie raportów stanu technicznego przed wydaniem pojazdu lub po jego zwrocie. Walidacja numeru VIN oraz roku produkcji (który nie może być ani w przyszłości, ani sprzed daty minimalnej dla danej marki) odbywa się już na etapie konstruktora.

`Offer` opisuje ofertę wynajmu. Przechowuje cennik dla różnych walut w kolekcji `ProductPrice`, datę ważności, przypisanie do działu (`VehicleDepartmentId` – konkretny pojazd lub `VehicleGroupId` – grupa pojazdów), limit kilometrów oraz domyślny depozyt. *Billing* to dokument rozliczeniowy zawierający pozycje `BillingItem`, kolekcję `Payment` i status cyklu życia (`DRAFT`, `ISSUED`, `PAID`, `CANCELED`), z osobnym statusem integracji ERP. `Payment` modeluje płatność poprzez metodę, kwotę w wybranej walucie, status i datę operacji. `Subscription` reprezentuje subskrypcję klienta platformy SaaS z planem cenowym, okresem rozliczeniowym oraz mechanizmem przedłużania. Ich cykl życia obsługują procesory warstwy aplikacyjnej.

Wszystkie te agregaty są połączone wspólnym API: konstruktor wymaga `tenantId` jako pierwszego argumentu, walidacja danych odbywa się przy tworzeniu i modyfikacji obiektów, a zmiany stanu o znaczeniu biznesowym generują zdarzenia domenowe.

7.1.2.4. Reguły biznesowe i zdarzenia domenowe

Kluczowe reguły biznesowe są zapisane bezpośrednio w klasach agregatów. Wynika to z założenia, że obiekt domenowy nie powinien istnieć w stanie sprzecznym z modelem biznesowym, niezależnie od kontekstu jego użycia. Reguły dzielą się na różne kategorie: walidacje niezmienników, reguły przejść stanów i inne specyficzne ograniczenia biznesowe.

Pierwszą kategorię stanowią walidacje niezmienników, które muszą być spełnione przy konstrukcji lub modyfikacji obiektu. Ponownie przykładem jest klasa `Email` opisana w *7.1.2.2*, która odrzuca puste i niepoprawne adresy, normalizując poprawną wartość do postaci lower-case. Klasa `Vehicle` waliduje numer VIN oraz rok produkcji pojazdu. `PhoneNumber` sprawdza format numeru telefonu, a TimeZoneId potwierdza istnienie strefy czasowej w systemowej bazie danych. Klasa `Address` wymaga jednoczesnej obecności linii adresu, kodu pocztowego, miasta i kraju – częściowy adres nie jest tworzony. Szczególną formą walidacji niezmiennika jest również metoda `Tenant.UpdateSubdomain`. Unikalność subdomeny i blokada nazw zarezerwowanych nie są egzekwowane na poziomie agregatu, ponieważ wymagają dostępu do bazy danych i są realizowane w wyższych warstwach aplikacji.

Drugą kategorię tworzą reguły przejść stanów, najlepiej widoczne w metodzie przedstawionej na *Listing 4*, czyli `Reservation.UpdateStatus`. Każde dopuszczalne przejście jest jawnie zakodowane warunkiem, a maszyna stanowa nie ukrywa zabronionych skoków - są one wyraźnie określone jako niedozwolone. Ta sama zasada dotyczy procesu wydania i zwrotu pojazdu, cyklu fakturowania oraz aktywacji/dezaktywacji subskrypcji.

Listing 4 Fragment klasy `Reservation` z projektu `Wellhicle.Domain` - metoda `UpdateStatus`. Źródło: opracowanie własne.

```
public void UpdateStatus(ReservationStatus status)
{
    if (Status == status)
        throw new InvalidOperationException("Reservation is already in
the specified status.");
    if (status == ReservationStatus.PENDING)
        throw new InvalidOperationException("Cannot revert reservation to
PENDING status.");
    if (status == ReservationStatus.CONFIRMED && Status !=
ReservationStatus.PENDING)
        throw new InvalidOperationException("Only PENDING reservations
can be moved to CONFIRMED status.");
    if (status == ReservationStatus.ACTIVE && Status !=
ReservationStatus.CONFIRMED)
        throw new InvalidOperationException("Only CONFIRMED reservations
can be activated.");
    if (status == ReservationStatus.CANCELED && (Status !=
ReservationStatus.PENDING && Status != ReservationStatus.CONFIRMED))
        throw new InvalidOperationException("Only PENDING or PLANNED
reservations can be canceled.");
    if (status == ReservationStatus.COMPLETED && Status !=
ReservationStatus.ACTIVE)
        throw new InvalidOperationException("Only ACTIVE reservations can
be completed.");
    if (status == ReservationStatus.NO_SHOW && Status !=
ReservationStatus.CONFIRMED)
        throw new InvalidOperationException("Only CONFIRMED reservations
can be marked as NO_SHOW.");
    if (status == ReservationStatus.INTERRUPTED && Status !=
ReservationStatus.ACTIVE)
        throw new InvalidOperationException("Only ACTIVE reservations can
be marked as INTERRUPTED.");
    Status = status;
}
```

Trzecią kategorię reguł stanowią zasady edytowalności. Rozdzielają one stany umożliwiające modyfikację obiektów od tych, w których są one wyłącznie do odczytu. Rezerwacja udostępnia w tym celuodpowiednie metody, np. `EnsureIsNotReadOnly`. Metody te muszą być wywoływane przed każdą operacją modyfikującą. Statyczna metoda `IsReadOnly(status, pickupStatus,`

returnStatus) określa warunki, w których rezerwacja staje się niemodyfikowalna - gdy nie znajduje się w stanach *PENDING* lub *CONFIRMED* albo zwrot został już zakończony. Stan pojazdu (*VehicleCondition*) blokuje zmiany raportu technicznego, jeśli proces jego generowania jest ukończony, a operacje rozliczeniowe w module *Billing* nie są dopuszczalne dla dokumentów już anulowanych.

Czwartą kategorię tworzą reguły spójności relacji między obiektami. Metoda *ValidateXorReference* w klasie *ProductPrice* gwarantuje, że cena jest powiązana dokładnie z jednym typem obiektu (plan subskrypcyjny, oferta lub dodatek), przy czym referencja XOR jest implementowana bezpośrednio w agregacie. *BillingItem* wymaga unikalnej i jednoznacznej referencji do konkretnego bytu biznesowego (rezerwacja, subskrypcja lub dodatek). Metoda *AddItem* w module *Billing* sprawdza przynależność nowej pozycji rozliczeniowej do odpowiedniego dokumentu, a metoda *AddGroup* dla klasy *Vehicle* odrzuca próby podwójnego przypisania tej samej grupy.

Szczególną kategorię reguł stanowią zdarzenia domenowe (domain events), które sygnalizują wystąpienie określonych faktów w modelu danych i są przetwarzane przez system [107]. W obszarze użytkowników klasa *User* generuje następujące zdarzenia: *UserRegisteredEvent* po zakończeniu rejestracji, *UserEmailUpdatedEvent* przy zmianie adresu e-mail (wymagające ponownej weryfikacji), *EmailVerifiedEvent* po potwierdzeniu nowego adresu oraz *PasswordChangedEvent* po aktualizacji hasła (*Listing 5*). Wzorzec rejestracji zdarzeń jest identyczny dla każdego typu: handler aplikacyjny wywołuje metodę domenową, która modyfikuje stan obiektu i zapisuje zdarzenie w kolekcji *_domainEvents* za pomocą chronionej metody *AddDomainEvent*.

Listing 5 Fragment klasy `User` *z projektu* `Wellhicle.Domain` *– metoda* `ChangePassword`*. Źródło: opracowanie własne.*

```
public void ChangePassword(string newPasswordHash)
{
    if (string.IsNullOrWhiteSpace(newPasswordHash))
        throw new ArgumentException("Password hash cannot be null or
empty.", nameof(newPasswordHash));

    PasswordHash = newPasswordHash;

    AddDomainEvent(new PasswordChangedEvent(Id, TenantId, Email));
}
```

Sama domena nie wykonuje operacji pobocznych. Nie wysyła e-maili, nie publikuje wiadomości na kolejce ani nie zapisuje audytu. Zapisuje jedynie fakt. Warstwa aplikacji odczytuje kolekcję *DomainEvents* po pomyślnym zapisie i przekazuje zdarzenia do publikacji. Dzięki temu reguła biznesowa „zmiana hasła wymaga powiadomienia użytkownika” nie wymusza na agregacie znajomości mechanizmu kolejkowania, a sama operacja domenowa pozostaje deterministyczna i jednostkowo testowalna.

Obiekty wartości pełnią w tej strukturze rolę uzupełniającą. Klasy *Email*, *PhoneNumber*, *Address*, *TimeZoneId*, *PersonalDocument* oraz *CompanyData* zamykają walidację i normalizację w typach, zamiast wymuszać je w handlerach. W rezultacie reguły biznesowe nie są rozsiane po warstwie aplikacyjnej. Pozostają tam, gdzie znajdują się odpowiadające im pojęcia modelu.

7.1.3. Warstwa aplikacji – wzorzec CQRS

Warstwa aplikacyjna projektu pośredniczy między światem domeny a światem prezentacji. To w niej decydowane jest, jakie operacje system udostępnia użytkownikom oraz w jakiej kolejności składane są reguły domenowe, aby te operacje doprowadzić do skutku.

7.1.3.1. Założenia CQRS i ich zastosowanie w projekcie

Segregacja odpowiedzialności poleceń i zapytań (ang. *Command Query Responsibility Segregation* - CQRS) to wzorzec architektoniczny, który rozdziela operacje zmieniające stan systemu od operacji służących wyłącznie do odczytu danych. Wprowadzony pierwotnie przez Grega Younga [108] jako rozszerzenie zasady rozdzielenia poleceń i zapytań Bertranda Meyera, został później szczegółowo omówiony przez Martina Fowlera [109] oraz w dokumentacji Microsoftu [110]. Wzorzec ten nie narzuca konieczności użycia dwóch oddzielnych baz danych. Możliwa jest jego uproszczona implementacja, gdzie zarówno operacje zapisu, jak i odczytu korzystają z tej samej bazy relacyjnej przy wykorzystaniu odrębnych modeli na poziomie warstwy aplikacyjnej [109].

W projekcie Wellhicle zastosowano właśnie tę uproszczoną wersję. Rozdzielenie operacji zapisu i odczytu jest realizowane poprzez organizację kodu na poziomie klas, interfejsów i typów aplikacyjnych bez konieczności modyfikacji infrastruktury bazodanowej. Oba komponenty korzystają z tej samej relacyjnej bazy danych zarządzanej przez Entity Framework Core za pośrednictwem wspólnego kontekstu danych. W praktyce oznacza to istnienie dwóch równoległych światów klas: po stronie zapisu znajdują się komendy wraz z ich handlerami, natomiast po stronie odczytu zapytania i ich handlery. Granica między nimi jest wyraźnie określona przez strukturę katalogów i interfejsów, nie schemat bazy danych.

Pełna implementacja CQRS z osobną bazą odczytową wymagałaby dodatkowej infrastruktury synchronizacyjnej oraz mechanizmów gwarantujących spójność ostateczną, co dla docelowej skali systemu Wellhicle byłoby nieuzasadnione operacyjnie. Zastosowany wariant zachowuje jednak kluczowe korzyści wzorca. Są to m.in. jasne rozgraniczenie odpowiedzialności poszczególnych komponentów, ułatwienie testowania pojedynczych przypadków użycia oraz możliwość niezależnej optymalizacji ścieżek odczytowych bez wpływu na model zapisu.

7.1.3.2. Komendy, zapytania i ich handlery

Każdy moduł biznesowy w warstwie aplikacyjnej dzieli swój kod na podkatalogi przeznaczone osobno dla komend i osobno dla zapytań. Komenda jest klasą danych opisującą intencję zmiany stanu. Zapytanie jest klasą reprezentującą żądanie pobrania danych. Same komendy i zapytania nie zawierają logiki biznesowej. Pełnią wyłącznie rolę nośników parametrów wejściowych. Logikę przetwarzania realizują handlery, czyli implementacje trzech kluczowych interfejsów stanowiących fundament architektury warstwy aplikacyjnej.

Listing 6 Interfejsy `ICommandHandler` oraz `IQueryHandler` z projektu `Wellhicle.Application`. Źródło: opracowanie własne.

```
namespace Wellhicle.Application.Abstractions;

public interface ICommandHandler<in TCommand, TResult> where TCommand :
class
{
    Task<Result<TResult>> HandleAsync(TCommand command, CancellationToken
cancellationToken = default);
}
```

```
public interface ICommandHandler<in TCommand> where TCommand : class
{
    Task<Result> HandleAsync(TCommand command, CancellationToken
cancellationToken = default);
}

public interface IQueryHandler<in TQuery, TResult> where TQuery : class
{
    Task<Result<TResult>> HandleAsync(TQuery query, CancellationToken
cancellationToken = default);
}
```

Wariant bezzwrotny `ICommandHandler<TCommand>` prezentowany na *Listing 6* służy do obsługi operacji, które nie zwracają danych do warstwy prezentacji (np. usuwanie). Pozostałe handlery komend zwracają typ wynikowy opisany w podpunkcie *7.1.3.3*. Handler komendy i odpowiadająca mu komenda znajdują się w tym samym pliku, co ułatwia lokalizację pary wiadomość-handler. Zależności są wstrzykiwane przez konstruktor, a warstwa prezentacji odwołuje się do handlerów wyłącznie poprzez te interfejsy.

Typowy handler komendy wykonuje cztery kroki w stałej kolejności: weryfikuje autoryzację, waliduje kontekst tenanta, sprawdza warunki biznesowe na pobranych z bazy obiektach, a następnie modyfikuje stan i potwierdza zapis. W przypadku operacji obejmujących wiele encji zapis jest opakowywany transakcją. Każdy nieprawidłowy krok kończy się natychmiastowym zwróceniem ujednoliconego wyniku z odpowiednim kodem błędu.

Najpełniej wzorzec widoczny jest w `CreateReservationCommandHandler` (*Listing 7*), który obejmuje wszystkie elementy mechanizmu w jednej operacji. Handler wstrzykuje kilkanaście współpracujących usług (np. kontekst żądania, autoryzację, walidację, fabryki obiektów wartości, czas systemowy, publikację powiadomień), lecz sam implementuje niemal wyłącznie walidacje wejściowe i kompozycję wyniku. Sekwencja obejmuje rozróżnienie ścieżki w zależności od typu konta (klient, pracownik, właściciel), pobranie i walidację klienta, oferty oraz pojazdu. Następnie, wewnątrz transakcji, utworzenie użycia pojazdu, konstrukcję encji rezerwacji w warstwie domeny, zapis dodatków oraz utworzenie dokumentu rozliczeniowego z pozycjami i płatnościami. Po zatwierdzeniu publikowane jest powiadomienie, a na końcu zwracany jest DTO.

Listing 7 Fragment metody `HandleAsync` z klasy `CreateReservationCommandHandler`, z projektu `Wellhicle.Application`. Źródło: opracowanie własne.

```
public async Task<Result<ReservationDto>>
HandleAsync(CreateReservationCommand command, CancellationToken
cancellationToken = default)
{
...
   try
    {
        await _unitOfWork.BeginTransactionAsync(cancellationToken);

        var vehicleUsageResult = await
_vehicleUsageService.CreateUsageAsync(/* ... */, cancellationToken);
        if (vehicleUsageResult.IsFailure) { await
_unitOfWork.RollbackTransactionAsync(cancellationToken); return /* ...
*/; }
        await _unitOfWork.SaveChangesAsync(cancellationToken);

        var reservation = new Reservation(tenantId, offer.Id, /* ... */);
        reservationRepository.Add(reservation);
        await _unitOfWork.SaveChangesAsync(cancellationToken);
```

```
        var billingResult = await
_billingService.CreateBillingAsync(tenantId, BillingType.SALE,
command.Billing, cancellationToken);
        var billingItemsResult = await
_billingService.AddBillingItemsAsync(tenantId, billingResult.Value!.Id,
billingItems, cancellationToken);
        var billingPaymentsResult = await
_billingService.AddPaymentsAsync(tenantId, billingResult.Value!.Id,
command.Billing.Payments.ToList(), cancellationToken);

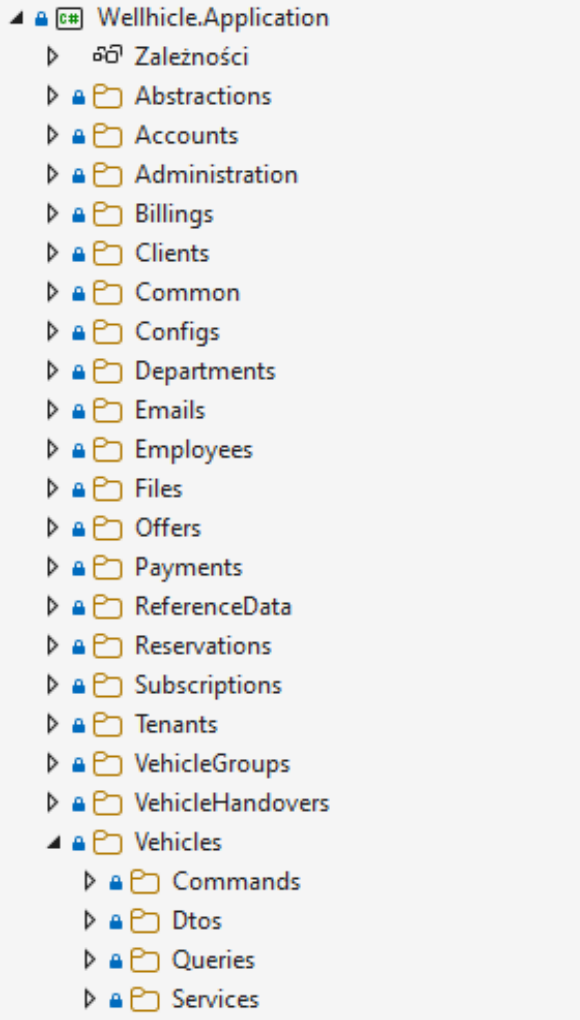
        await _unitOfWork.CommitTransactionAsync(cancellationToken);
 ...
   }
    catch (Exception)
    {
        await _unitOfWork.RollbackTransactionAsync(cancellationToken);
        return Result<ReservationDto>.Failure("An error occurred while
creating the reservation", ErrorCode.GENERIC_ERROR);
    }
}
```

Handler pełni rolę dyrygenta, podczas gdy logikę szczegółową realizują inne wstrzykiwane usługi opisane w podpunkcie *7.1.3.4*. Zapis fizyczny jest wykonywany przez warstwę persystencji, a reguły biznesowe sprawdzalne w obrębie pojedynczego agregatu (niezmienniki konstrukcji, przejścia stanów, edytowalność) pozostają zaimplementowane bezpośrednio w encjach domenowych (zgodnie z opisem w podpunkcie *7.1.2.4*). Reguły wymagające stanu innych agregatów, takie jak sprawdzenie, czy pojazd jest wolny w żądanym przedziale czasowym, czy adres e-mail jest unikalny w skali tenanta lub czy subdomena nie jest już zajęta, są implementowane w handlerze lub w wyspecjalizowanych usługach aplikacyjnych (np. IVehicleUsageService), ponieważ agregat z definicji nie sięga poza swoją granicę. Sam handler nie posiada wiedzy o szczegółach implementacji Entity Framework, strukturze bazy danych ani kontekście HTTP.

Handlery zapytań działają na podobnej zasadzie, ale nigdy nie modyfikują stanu trwałego i nie inicjują transakcji. W prostszych przypadkach pobierają dane za pomocą generycznego repozytorium z projekcją, natomiast w bardziej złożonych scenariuszach wykorzystują wyspecjalizowane komponenty odczytowe. Te ostatnie łączą zapytanie z mechanizmami kontroli dostępu oraz obliczają pola pochodne na poziomie SQL, unikając materializacji pełnych grafów encji. Wynik jest następnie mapowany na DTO poprzez metody rozszerzające zdefiniowane w encjach, a dla zapytań listowych pakowany w strukturę stronicowaną. Wybór ścieżki odczytu zależy od stopnia skomplikowania reguł widoczności obowiązujących w danym module.

7.1.3.3. Wspólne kontrakty warstwy aplikacyjnej

Warstwa aplikacyjna jest fizycznie wydzielona jako osobny projekt .NET i wewnętrznie zorganizowana wokół dwóch typów katalogów: wspólnych (*Abstractions*, zawierających interfejsy implementowane w warstwie infrastruktury oraz *Common*, obejmujących typy używane przekrojowo) oraz katalogów odpowiadających obszarom biznesowym aplikacji. Dokładny podział przedstawiony jest na *Rysunek 96 Struktura projektu Wellhicle.Application w Visual Studio. Źródło: opracowanie* własne. Wewnątrz każdego modułu biznesowego utrzymywany jest powtarzalny układ podkatalogów (*Commands*, *Queries*, *Dtos*, *Services*), choć w modułach o węższym zakresie niektóre z nich mogą nie występować.



Rysunek 96 Struktura projektu Wellhicle.Application w Visual Studio. Źródło: opracowanie własne.

Komunikacja między handlerami a warstwą prezentacji opiera się na czterech wspólnych kontraktach, stosowanych jednolicie w całym projekcie. Pierwszy z nich stanowią obiekty transferu danych (ang. *Data Transfer Object* – DTO), służące do przenoszenia informacji między warstwami systemu w pojedynczym wywołaniu, bez implementacji logiki biznesowej [111]. DTO dzielą się na wyjściowe (zakończone przyrostkiem `Dto` lub `ListDto`), reprezentujące odpowiedzi handlerów, oraz wejściowe (oznaczone prefiksem `Set`), opisujące strukturę żądania. Obiekty te często zawierają pola wyliczane w warstwie aplikacji. Przykładowo, status edytowalności zależny od roli użytkownika lub informacje o konfliktach generowane przez serwis – których nie ma w encjach persystencji, a także typy danych lepiej dopasowane do API (np. daty z jawnym przesunięciem strefy czasowej zamiast czasu UTC).

Drugim kontraktem jest ujednolicony typ wynikowy `Result<T>` (*Listing 8*). Wykorzystywany jest jako zwrot ze wszystkich handlerów oraz znacznej części usług aplikacyjnych. Klasa oferuje trzy warianty fabryczne dla obsługi błędów: z dowolnym komunikatem, z komunikatem i kodem błędu lub wyłącznie z kodem (w tym ostatnim przypadku komunikat jest pobierany ze wspólnego słownika domyślnych wiadomości).

Listing 8 Klasa `Result` z projektu `Wellhicle.Application`. Źródło: opracowanie własne.

```
namespace Wellhicle.Application.Common;

public class Result<T>
{
    public T? Value { get; private set; }
```

```
    public bool IsSuccess { get; private set; }
    public bool IsFailure => !IsSuccess;
    public string? ErrorMessage { get; private set; }
    public string? ErrorCode { get; private set; }

    public static Result<T> Success(T value) => new(value, true);
    public static Result<T> Failure(string errorMessage, string?
errorCode = null) => new(default, false, errorMessage, errorCode);
    public static Result<T> Failure(string errorMessage, ErrorCode
errorCode) => new(default, false, errorMessage,
((int)errorCode).ToString());
    public static Result<T> Failure(ErrorCode errorCode) => new(default,
false, ErrorMessages.GetMessage(errorCode), ((int)errorCode).ToString());
}
```

Trzecim kontraktem jest enum kodów błędów. Zawiera ponad sto siedemdziesiąt wartości pogrupowanych tematycznie, z których każda ma jawnie przypisany numer wystawiany w odpowiedziach API. Domyślne komunikaty są przechowywane we wspólnym słowniku, a część handlerów uzupełnia je o lokalne komunikaty lepiej oddające kontekst biznesowy. Mechanizm ten jest dodatkowo rozszerzony o mapowanie na kody HTTP oraz zbiór kodów cenzurowanych – wartości, których szczegóły nie powinny być ujawniane na zewnątrz (np. błędy logowania, identyfikacji konta lub awarie infrastruktury). Dzięki temu warstwa prezentacyjna nie różnicuje między komunikatami takimi jak „błędne hasło” a „nieistniejące konto”.

Czwartym kontraktem jest stronicowanie. Parametry stronicowania i sortowania są przekazywane w jednym typie wejściowym, który zawiera domyślne oraz maksymalne rozmiary strony, a także metodę normalizującą wartości spoza zakresu. Wynik zapytań listowych jest pakowany w typ stronicowany, zawierający pola wyliczeniowe (`TotalPages`, `HasNextPage`, `HasPreviousPage`) oraz niegeneryczny znacznik rozpoznawany przez filtry akcji w warstwie prezentacji. Dzięki tym czterem kontraktom kontroler obsługujący dowolny obszar biznesowy zawsze otrzymuje odpowiedź o spójnej strukturze.

7.1.3.4. Usługi aplikacyjne i konteksty wykonania

Obok handlerów warstwa aplikacyjna zawiera usługi aplikacyjne, których zadaniem jest kapsułkowanie powtarzalnej logiki przypadków użycia oraz orkiestracja kroków, których handler nie powinien wykonywać samodzielnie. Pełnią one cztery wyraźne role. Po pierwsze, usługi kontekstu i kontroli dostępu sprawdzają autoryzację, ustalają i weryfikują kontekst tenanta oraz rozwiązują pytanie „do którego działu należy aktualny pracownik”. Bez nich każdy handler musiałby samodzielnie implementować kilka linii kodu odpowiedzialnych za sklejanie kontekstu żądania, repozytoriów użytkowników i sprawdzanie aktywności tenanta. Po drugie, pomocnicy modułowi zbierają walidacje wykorzystywane w więcej niż jednym handlerze (na przykład walidacja dodatków oferty wspólna dla tworzenia i aktualizacji rezerwacji). Po trzecie, koordynatorzy agregujących operacji opakowują wieloetapowe zapisy w pojedyncze metody. Jednym z takich przykładów jest `IBillingService`, który udostępnia kilkanaście operacji na rozliczeniach, pozycjach i płatnościach, a `IVehicleUsageService` rozwiązuje walidację konfliktów dat i zakresów przypisań pojazdu w jednym wywołaniu. Po czwarte, usługi techniczno-aplikacyjne odpowiadają za zagadnienia przekrojowe: wielojęzyczność, tryb podglądu rozliczeń oraz fabryki obiektów wartości enkapsulujące walidację i tworzenie typów domenowych.

Drugą grupę elementów wspierających stanowią konteksty wykonania – klasy dostarczające informacji o otoczeniu operacji, ale nie wykonujące pracy biznesowej. Najważniejszy z nich widoczny

na *Listing 9* opisuje aktualnie zalogowanego użytkownika i jest wstrzykiwany w niemal każdym handlerze.

Listing 9 Interfejs `IUserContext` z projektu `Wellhicle.Application`. Źródło: opracowanie własne.

```
namespace Wellhicle.Application.Abstractions;

public interface IUserContext
{
    int? UserId { get; }
    int TenantId { get; }
    string? Email { get; }
    AccountType AccountType { get; }
    bool IsAuthenticated { get; }
}
```

Wartości pól pochodzące z `ClaimsPrincipal`, ustawianego przez warstwę uwierzytelniającą, są wykorzystywane przez handlery do rozróżniania ścieżek przetwarzania żądań w zależności od roli oraz egzekwowania ograniczeń widoczności. Oznacza to na przykład, że klient może odczytać tylko swój profil, niezależnie od identyfikatora podanego w żądaniu. Kontekst tenanta jest rozdzielony na dwie niezależne funkcjonalności: rozwiązanie identyfikatora tenanta z hosta żądania oraz udostępnienie tej wartości podczas obsługi żądania. Czas systemowy reprezentowany jest przez osobny kontrakt, `IClock`, zawierający dwa pola – bieżącą datę UTC oraz aktualną datę, co umożliwia zastąpienie go atrapą w testach bez konieczności wstrzykiwania czasu do każdej klasy.

Trzecią grupę stanowią dane referencyjne i komponenty rozwiązujące (ang. *resolver*): wąskie interfejsy udostępniające szybki odczyt rekordów słownikowych na podstawie kodu ISO oraz cienkie warstwy wykorzystujące te mechanizmy wyszukiwania z domyślnymi wartościami tenanta jako rozwiązanie awaryjne. Prezentacja danych referencyjnych znajduje się w osobnym module o nietypowej strukturze. Zawiera ona wyłącznie query, DTO i serwisy, bez komend, ponieważ jego rola ogranicza się do wystawienia danych słownikowych pod punkty końcowe API.

Wszystkie te elementy łączy ta sama zasada. Handler wywołuje wyłącznie interfejsy zdefiniowane w warstwie aplikacyjnej, nie znając konkretnych implementacji infrastruktury. Dzięki temu warstwa Wellhicle.Application realizuje przypadki użycia bez wiedzy o HTTP, schemacie bazy danych czy kolejce zdarzeń. Wszystkie te szczegóły są ukryte za kontraktami, których implementacje pozostają wymienne.

7.1.4. Warstwa infrastruktury – dostęp do danych

Warstwa infrastruktury pełni rolę adaptera między abstrakcjami warstwy aplikacyjnej a fizycznym schematem relacyjnej bazy danych, ukrywając wszelkie szczegóły implementacyjne przed kodem realizującym logikę biznesową.

7.1.4.1. Repozytoria, kontekst danych i Unit of Work

Warstwa infrastruktury implementuje dostęp do danych przy użyciu Entity Framework Core, czyli biblioteki ORM rekomendowanej przez Microsoft jako domyślne rozwiązanie persystencji dla aplikacji .NET [112]. Cały kod związany z bazą danych znajduje się w katalogu `Wellhicle.Infrastructure/Persistence` i jest zorganizowany wokół trzech podstawowych elementów: pojedynczego kontekstu bazy danych, generycznego repozytorium oraz jednostki pracy. Obok nich w tym samym katalogu znajdują się konfiguracje encji w Fluent API, interceptory operacji na plikach, migracje oraz wyspecjalizowane repozytoria odczytowe omawiane w podpunkcie *7.1.4.3*.

Kontekst bazy danych `AppDbContext` dziedziczy po standardowym `DbContext` z EF Core i udostępnia kolekcje encji (`DbSet<T>`) dla wszystkich agregatów domeny, które wymagają utrwalenia. Konstruktor przyjmuje dwa opcjonalne komponenty pochodzące z warstwy aplikacyjnej: identyfikator aktualnego tenanta oraz zegar systemowy. Oba są wykorzystywane w nadpisanej metodzie `SaveChangesAsync` kontekstu – pierwszy do automatycznego przypisywania wartości `TenantId` nowym encjom typu `ITenantOwned`, drugi do uzupełniania pól audytowych dla encji typu `IAuditable`. Mechanizm ten został szczegółowo opisany w sekcji *7.1.4.2*. Decyzja o opcjonalności obu komponentów wynika z konieczności obsługi scenariuszy, w których kontekst jest tworzony poza przetwarzaniem żądań HTTP (np. podczas migracji lub testów jednostkowych).

Generyczne repozytorium (`Repository<T>`) implementuje kontrakt `IRepository<T>` zdefiniowany w warstwie aplikacyjnej i opakowuje `DbSet<T>` pobierany z kontekstu. Klasa udostępnia dwie rodziny metod o symetrycznych nazwach, ale różnym zakresie działania. Pierwsza rodzina (z przyrostkiem *InTenantAsync*) działa wyłącznie w obrębie aktualnego tenanta i jest używana niemal w każdym handlerze. Automatycznie ogranicza wynik do rekordów należących do bieżącego klienta. Druga rodzina (z przyrostkiem *GlobalAsync*) wykonuje operacje bez tego filtra i jest stosowana w przypadkach między-tenantowych: sprawdzania unikalności adresu e-mail w skali całej platformy, dostępu do encji globalnych czy operacji administracyjnych. Większość metod obu rodzin korzysta z tych samych operacji Entity Framework Core – projekcji LINQ, predykatów oraz ładowania powiązanych encji przez `Include`. Różni się jedynie obecnością albo brakiem warunku filtrującego po `TenantId`. Wyjątkiem jest pobieranie po identyfikatorze, gdzie wariant tenantowy używa `DbSet.FindAsync` z weryfikacją przynależności do tenanta w pamięci, a wariant globalny predykatu SQL na kolumnie `Id`.

Jednostka pracy (ang. *Unit of Work*) prezentowana na *Listing 10* stanowi pojedynczy punkt dostępu warstwy aplikacyjnej do mechanizmów persystencji. Pełni dwie kluczowe funkcje: udostępnia repozytoria dla dowolnego typu encji oraz zarządza transakcjami bazy danych. Zgodnie z definicją wzorca projektowego Unit of Work, jego rola polega na utrzymywaniu listy obiektów objętych transakcją biznesową, koordynacji zapisu zmian oraz rozwiązywaniu problemów związanych ze współbieżnością [113]. W projekcie model ten został rozszerzony o mechanizm leniwej fabryki repozytoriów: podczas pierwszego odczytu repozytorium dla konkretnego typu encji tworzona jest jego instancja, natomiast kolejne wywołania zwracają tę samą instancję przechowywaną w słowniku w obrębie bieżącej jednostki pracy. Dzięki temu wszystkie operacje wykonywane w obrębie pojedynczego żądania dzielą wspólny kontekst zmian.

Listing 10 Fragment implementacji klasy `UnitOfWork` z projektu `Wellhicle.Infrastructure`. Źródło: Opracowanie własne.

```
namespace Wellhicle.Infrastructure.Persistence;

public class UnitOfWork : IUnitOfWork
{
private readonly AppDbContext _context;
private readonly Dictionary<Type, object> _repositories = new();

public UnitOfWork(AppDbContext context, /* ... */)
{
    _context = context;
}

public IRepository<T> GetRepository<T>() where T : class
{
    var type = typeof(T);
    if (!_repositories.ContainsKey(type))
    {
        _repositories[type] = new Repository<T>(_context);
```

```
    }
    return (IRepository<T>)_repositories[type];
}

public async Task BeginTransactionAsync(CancellationToken
cancellationToken = default)
    => await _context.BeginTransactionAsync(cancellationToken);

public async Task CommitTransactionAsync(CancellationToken
cancellationToken = default)
    => await _context.CommitTransactionAsync(cancellationToken);

public async Task RollbackTransactionAsync(CancellationToken
cancellationToken = default)
    => await _context.RollbackTransactionAsync(cancellationToken);
}
```

Transakcje są obsługiwane przez trzy metody delegujące do kontekstu: rozpoczynającą transakcję, zatwierdzającą oraz cofającą. Handler komendy używa ich w jawnym schemacie try–commit–catch–rollback, gdy operacja obejmuje zapis wielu encji, których spójność musi być utrzymana w jednym kroku (przykładem jest opisany w *7.1.3.2* handler tworzenia rezerwacji). Operacje prostsze, sprowadzające się do pojedynczego `SaveChanges`, nie wymagają jawnej transakcji, ponieważ EF Core domyślnie opakowuje każdy zapis własną transakcją na poziomie bazy.

Zysk wynikający z tego układu jest wyraźnie odczuwalny w warstwie aplikacyjnej. Handler nie zna typu konkretnej bazy danych ani szczegółów EF Core, a operuje wyłącznie na trzech kontraktach z `Wellhicle.Application/Abstractions`: jednostce pracy, repozytorium generycznym i opcjonalnie repozytoriach odczytowych. Po stronie infrastruktury każda z tych abstrakcji ma dokładnie jedną implementację, co eliminuje konieczność rejestrowania złożonych map zależności w kontenerze wstrzykiwania zależności. Tak rozumiana granica między warstwami pozwala wymienić mechanizm persystencji bez zmian w logice biznesowej. Koszt potencjalnej migracji do innego dostawcy bazy danych sprowadza się do reimplementacji trzech klas w jednym katalogu.

7.1.4.2. Mechanizmy automatyczne w cyklu zapisu

Wokół pomyślnego zapisu w bazie danych dzieją się dwa procesy, których handler aplikacyjny nie wywołuje bezpośrednio, ale na których musi polegać. Pierwszy z nich automatycznie uzupełnia pola, takie jak identyfikator tenanta oraz znaczniki audytowe (`CreatedAt`/`UpdatedAt`), co eliminuje konieczność ręcznego aktualizowania tych wartości przy każdej operacji. Drugi zapewnia propagację zdarzeń domenowych poza warstwę persystencji, gdy zapis jest udany. Wspólną cechą obu procesów jest ich automatyczne wykonanie w momencie wywołania metody `SaveChanges`, bez potrzeby dodatkowego angażowania warstwy aplikacyjnej.

Pierwszy mechanizm to nadpisana metoda `SaveChangesAsync` w kontekście danych. Przed zapisem przechodzi ona przez wszystkie śledzone encje i wykonuje dwie operacje uzupełniające. Encje typu `ITenantOwned` (szczegółowo opisane w sekcji *7.1.2.2*) znajdujące się w stanie `Added`, dla których właściwość `TenantId` ma wartość zero (tj. nie została ustawiona w konstruktorze), otrzymują identyfikator bieżącego tenanta pobrany z `ITenantProvider`. To rozwiązanie eliminuje ryzyko wystąpienia błędów wynikających z pominięcia przekazania wartości `TenantId` do agregatu utworzonego w handlerze. Encje implementujące interfejs `IAuditable` (szczegółowo opisane w sekcji *7.1.2.2*) są automatycznie uzupełniane o pola audytowe: `CreatedAt` przy tworzeniu nowej encji oraz `UpdatedAt` przy każdej modyfikacji. W przypadku operacji modyfikujących zmiana wartości `CreatedAt` jest jawnie blokowana, co zapobiega przypadkowemu nadpisaniu pola „data utworzenia” późniejszymi zapisami. Czas jest pobierany z `IClock` (szczegółowo opisany w sekcji

7.1.3.4) lub bezpośrednio z `DateTime.UtcNow`, jeżeli kontekst nie został utworzony w pełnym środowisku aplikacji.

Listing 11 Metoda `PublishDomainEventsAsync` z klasy `UnitOfWork`, z projektu `Wellhicle.Infrastructure`. Źródło: opracowanie własne.

```
private async Task PublishDomainEventsAsync(CancellationToken
cancellationToken)
{
    var entitiesWithEvents = _context.ChangeTracker.Entries()
        .Where(e => e.Entity is AggregateRoot aggregateRoot &&
aggregateRoot.DomainEvents.Any())
        .Select(e => (AggregateRoot)e.Entity)
        .ToList();

    foreach (var entity in entitiesWithEvents)
    {
        var events = entity.DomainEvents.ToList();
        foreach (var domainEvent in events)
        {
            try
            {
                await _domainEventPublisher!.PublishAsync(domainEvent,
cancellationToken);
            }
            catch (Exception ex)
            {
                System.Diagnostics.Debug.WriteLine($"Błąd publikacji
zdarzenia: {ex.Message}");
            }
        }

        entity.ClearDomainEvents();
    }
}
```

Publikacja zdarzenia widoczna na *Listing 11* jest otoczona blokiem try–catch, który zapobiega przerwaniu całej operacji przez niepowodzenie pojedynczego zdarzenia. Błąd zostaje jedynie odnotowany, a kolejne zdarzenia są przetwarzane normalnie i trafiają do kolejki. Obecna implementacja stanowi świadomy kompromis: gwarantuje, że pomyślny commit w bazie danych nie zostanie cofnięty na skutek chwilowej awarii systemu kolejkowania wiadomości. Dla operacji wymagających wyższego poziomu gwarancji dostarczenia zdarzeń (atomowość zapisu i publikacji) zaleca się implementację wzorca outbox [114] jako rozwiązanie docelowe. W obecnej fazie projektu nie jest on jednak uwzględniony.

Oba mechanizmy współdziałają, realizując zasadę, że warstwa aplikacyjna nie powinna ręcznie obsługiwać typowych zagadnień występujących w niemal każdej operacji. Handler tworzący rezerwację nie zapisuje pól audytowych ani nie wywołuje publishera zdarzeń. Te czynności są realizowane automatycznie po wykonaniu metody `SaveChangesAsync`. `TenantId` handler ustawia jawnie przez konstruktor agregatu. Mechanizm automatyczny w `SaveChangesAsync` służy tu jako siatka bezpieczeństwa dla przypadków brzegowych, w których w konstruktorze przekazano wartość 0. Dostęp do danych dla konkretnego tenanta jest zabezpieczony na dwóch poziomach: poprzez rodzinę metod `InTenantAsync` w generycznym repozytorium (szczegółowy opis znajduje się w podpunkcie *7.1.4.1*) oraz przez jawną filtrację danych w repozytoriach odczytowych (opisaną szczegółowo w podpunkcie *7.1.4.3*). Sposób przetwarzania zdarzeń publikowanych przez `IDomainEventPublisher`, takich jak mechanizmy kolejkowania, czy obsługa po stronie `EmailWorker` jest opisany w podpunkcie *7.1.8.2*.

7.1.4.3. Repozytoria odczytowe i optymalizacje SQL

Generyczne `Repository<T>`, opisane w sekcji *7.1.4.1*, obsługuje większość operacji odczytu w projekcie jako proste pobrania po identyfikatorze, filtrowanie oraz paginację dla jednej encji. Jednak część zapytań wykracza poza ten model. Wymaga łączenia wielu encji, stosowania reguł widoczności opartych na roli użytkownika lub obliczania pól nieuwzględnionych bezpośrednio w schemacie bazy. W takich przypadkach warstwa infrastruktury udostępnia dedykowane repozytoria odczytowe: `IVehicleAccessReadRepository` oraz `IVehicleUsageReadRepository`. Repozytoria te są wstrzykiwane przez `UnitOfWork` na podobnej zasadzie jak generyczne repozytorium, jednak każda ich metoda jest implementowana ręcznie dla konkretnego zapytania.

W klasach tych wykorzystano powtarzalne techniki Entity Framework Core. Wszystkie zapytania odczytowe stosują opcję `AsNoTracking`, co uniemożliwia dodanie wyniku do kolekcji `ChangeTracker`. Dzięki temu eliminowany jest narzut związany z porównywaniem wartości i wykluczone ryzyko nieintencjonalnego zapisu. Izolacja tenanta realizowana jest poprzez jawny predykat `Where(v => v.TenantId == tenantId)`, gdzie każda metoda repozytorium odczytowego przyjmuje parametr `tenantId` i bezpośrednio włącza go do zapytania, zapewniając pełną kontrolę nad widocznością rekordów nawet w przypadku złożonych reguł dostępu opartych na roli pracownika i jego przynależności działowej. Projekcja LINQ jest kierowana bezpośrednio do typu wynikowego (przykład. Entity Framework Core tłumaczy całą operację na pojedyncze zapytanie SQL, zawierające wyłącznie niezbędne kolumny.

Najbardziej charakterystyczną cechą tych repozytoriów jest wyliczanie pól pochodnych na poziomie bazy danych. Status pojazdu nie jest bezpośrednio przechowywany w kolumnie, lecz wynika z aktualnego przypisania do działu, dat tego przypisania oraz typu aktywnego użycia pojazdu w danym momencie. Naiwna implementacja polegałaby na pobraniu wszystkich potrzebnych encji do pamięci i obliczeniu statusu w języku C#, co dla dużych list pojazdów oznaczałoby ładowanie kilku tabel jednocześnie, a następnie odrzucenie większości wyników. Projekcja rozwiązuje ten problem, zapisując logikę statusu jako serię warunków LINQ, które Entity Framework Core tłumaczy na konstrukcję `CASE WHEN` po stronie SQL (*Listing 12*). Wszystkie warunki muszą być wyrażone za pomocą operatorów trójargumentowych i porównań. Wywołanie metody nieobsługiwanej przez tłumacz LINQ-to-SQL przerwałoby proces translacji zapytania, wymuszając ewaluację po stronie klienta. Dzięki temu obliczenie statusu, mimo złożoności logicznej, sprowadza się do wykonania jednego zapytania SQL bez konieczności dodatkowych round-tripów do bazy danych.

Listing 12 Metoda `ProjectWithStatus` z klasy `VehicleAccessReadRepository`, z projektu `Wellhicle.Infrastructure`. Źródło: opracowanie własne.

```
private static IQueryable<VehicleDetails> ProjectWithStatus(
    IQueryable<VehicleAccessRow> query,
    DateOnly today)
{
    return query.Select(x => new VehicleDetails(
        x.Vehicle,
        x.Assignment,
        x.Assignment.DateFrom <= today && (x.Assignment.DateTo == null ||
x.Assignment.DateTo >= today)
            ? (x.ActiveUsageType == VehicleUsageType.SERVICE
                ? VehicleStatus.IN_SERVICE
                : x.ActiveUsageType == VehicleUsageType.RESERVATION
                    ? VehicleStatus.IN_RENTAL
                    : x.ActiveUsageType == VehicleUsageType.MANUAL_BLOCK
                        ? VehicleStatus.BLOCKED
                        : VehicleStatus.ACTIVE)
            : VehicleStatus.INACTIVE,
        x.ActiveUsageType));
```

```
}
```

Stronicowanie w tych repozytoriach realizowane jest na poziomie SQL, gdzie każda metoda paginacji wykonuje dwa zapytania: pierwsze to `COUNT(*)` z pełnym predykatem, a drugie to właściwy `SELECT` z klauzulą `ORDER BY` i `OFFSET ... FETCH NEXT`. Łączna liczba rekordów oraz dane dla konkretnej strony są przekazywane jako obiekt `PagedResult<T>`, jak opisano w podpunkcie *7.1.3.3*. Filtry dotyczące pól wyliczanych, np. „pokaż tylko pojazdy o statusie `IN_RENTAL`", są konwertowane na predykaty działające zgodnie z tymi samymi warunkami co projekcja, dzięki czemu operacje te odbywają się po stronie bazy. Decyzja o implementacji tych zapytań jako osobnych klas (a nie metod w repozytorium generycznym) wynika z konieczności łączenia kilku encji, jawnej filtracji widoczności oraz budowy projekcji, których nie da się wyrazić za pomocą interfejsu operującego na pojedynczym typie. Dotyczy to głównie dwóch obszarów o najbardziej rozbudowanym modelu dostępu: pojazdów i ich użyć. Pozostałe moduły systemu mogą korzystać z generycznego interfejsu `IRepository<T>`, który w pełni pokrywa ich zapotrzebowanie bez potrzeby stosowania wyspecjalizowanych repozytoriów.

7.1.5. Zabezpieczenia i autoryzacja

System Wellhicle opiera uwierzytelnianie i autoryzację na standardowych mechanizmach ASP.NET Core, takich jak tokeny JWT oraz hashowanie haseł [115] [116]. Kontekst użytkownika jest wstrzykiwany do warstwy aplikacyjnej. Konfiguracja tych komponentów nie odbiega znacząco od typowej aplikacji webowej wykorzystującej tę technologię, jednak kluczowe decyzje projektowe wpływają na sposób rozróżniania użytkowników i kontroli ich dostępu. Należą do nich: wielofrontendowa autoryzacja przy logowaniu, kaskada sprawdzeń przed wystawieniem tokenu oraz zasada, że autoryzacja odbywa się w handlerze (w warstwie aplikacyjnej), a nie w warstwie prezentacji.

7.1.5.1. Uwierzytelnianie i sesja użytkownika

Logowanie jest obsługiwane przez komendę `LoginCommand` oraz jej handler. Komenda przekazuje trzy pola: adres e-mail, hasło oraz nagłówek `X-Frontend-Source` o typie `LoginFrontendSource`. Pierwsze dwa pola są standardowe, natomiast trzecie wymaga uzasadnienia. System umożliwia logowanie za pośrednictwem trzech aplikacji frontendowych: `TenantFrontend` (Aplikacja do zarządzania wypożyczalnią), `CustomerFrontend` (Aplikacja do korzystania z usług wypożyczalni), `AdminFrontend` (Aplikacja do administrowania wypożyczalniami) oraz `OperationalMobile` (Aplikacja mobilna). Każda z tych aplikacji powinna umożliwiać logowanie wyłącznie określonym typom kont. Decyzja o tym, które konta mogą być używane w danej aplikacji, jest jawnie zakodowana za pomocą instrukcji switch przedstawionych w *Listing 13*.

Listing 13 Metoda `IsUserAllowedForFrontend` z klasy `LoginCommand`, z projektu `Wellhicle.Application`. Źródło: opracowanie własne.

```
private static bool IsUserAllowedForFrontend(AccountType accountType,
LoginFrontendSource frontendSource)
{
    return frontendSource switch
    {
        LoginFrontendSource.TenantFrontend => accountType is
AccountType.OWNER or AccountType.OFFICE or AccountType.OPERATIONAL,
        LoginFrontendSource.CustomerFrontend => accountType ==
AccountType.CLIENT,
        LoginFrontendSource.AdminFrontend => accountType ==
AccountType.ADMIN,
```

```
        LoginFrontendSource.OperationalMobile => accountType is
AccountType.OWNER or AccountType.OPERATIONAL,
        _ => false
    };
}
```

Bez tego rozróżnienia klient mógłby technicznie zalogować się do interfejsu panelu właściciela przy użyciu tych samych poświadczeń (choć nie mógłby wykonać żadnych akcji ze względu na brak odpowiednich ról). Dzięki nagłówkowi punkt końcowy logowania staje się świadomy kontekstu aplikacji i natychmiast odrzuca niedopuszczalne kombinacje połączeń roli użytkownika z docelowym panelem.

Proces uwierzytelnienia w systemie Wellhicle ma charakter kaskadowy. Handler kolejno sprawdza: poprawność formatu adresu e-mail (przy użyciu fabryki zwracającej `Result<Email>`), istnienie użytkownika oraz poprawność hasła (z wykorzystaniem interfejsu hashowania), dopuszczalność typu konta dla frontendu, stan tenanta i aktywność konta. Weryfikuje również przypisanie do aktywnego działu w przypadku pracowników oraz, jeśli jest to wymagane konfiguracją `IJwtFactory.RequireEmailVerification`, weryfikację adresu e-mail. Większość nieprawidłowych weryfikacji kończy się zwróceniem Result.Failure z konkretnym kodem błędu, z wyjątkiem dwóch przypadków (nieistniejące konto lub nieprawidłowe hasło oraz niewłaściwy typ konta dla frontendu), które zawsze zwracają ten sam kod `INVALID_EMAIL_OR_PASSWORD`. Jest to celowe maskowanie: API nie ujawnia, czy błąd wynika z nieprawidłowego hasła, braku konta czy próby logowania się klientem na panel właściciela. Dodatkowo, kod błędu jest filtrowany na poziomie `ErrorMessages` (szczegółowo opisanych w podpunkcie *7.1.3.3*), dzięki czemu nawet komunikat tekstowy pozostaje jednolity.

Pomyślne przejście kaskady uwierzytelniania kończy się wystawieniem dwóch tokenów: krótkoterminowego access tokenu (JWT z tożsamością użytkownika, identyfikatorem tenanta oraz typem konta) i refresh tokenu wykorzystywanego do odnowienia sesji bez konieczności ponownego podawania hasła. Wartości te są pakowane w strukturę `AuthResultDto`. Token JWT zawiera identyfikator użytkownika, identyfikator tenanta, adres e-mail oraz typ konta. Są to dane osadzane przez warstwę prezentacji i udostępniane handlerom za pośrednictwem `IUserContext` (szczegóły w sekcji *7.1.3.4*). Mechanizm umożliwia również unieważnienie wydanego tokenu (przez dodanie identyfikatora JTI do czarnej listy), co jest wykorzystywane podczas operacji wylogowania lub zmiany hasła.

Dla kont typu `OWNER` przewidziano odstępstwo od standardowej kaskady uwierzytelnienia. Właściciel może zalogować się do systemu nawet wtedy, gdy jego tenant jest nieaktywny. Jest to zabezpieczenie wynikające z procesu odnawiania subskrypcji. Właściciel musi mieć możliwość dostępu do panelu administracyjnego, aby ponownie aktywować tenanta. Dla pozostałych typów kont nieaktywny tenant stanowi twardą blokadę logowania.

7.1.5.2. Autoryzacja przypadków użycia i izolacja kontekstu

Kontrola dostępu do operacji jest realizowana na dwóch poziomach. Atrybuty `[Authorize]` na kontrolerach pełnią rolę pierwszego filtra, wymuszając uwierzytelnienie użytkownika oraz posiadanie odpowiedniego typu konta dla danego punktu końcowego przed przekazaniem żądania do handlera. Drugą warstwę stanowi interfejs `IAuthorizationService`, wstrzykiwany do handlerów i wykorzystywany do szczegółowych sprawdzeń wewnątrz przypadków użycia. Interfejs ten udostępnia rodzinę metod *Require** (np. `RequireAuthenticatedUser`, `RequireOwner`, `RequireClient`) oraz wariant `RequireAnyAccountType`, przyjmujący listę dopuszczalnych typów kont w dwóch wersjach: generycznej i niegenerycznej, dostosowanej do typu zwrotnego

handlera. Każda metoda zwraca wynik Result z odpowiednim kodem błędu (`UNAUTHENTICATED` lub `FORBIDDEN`), a handler propaguje ten błąd w sposób analogiczny do innych błędów walidacji.

Wybór dodatkowej warstwy autoryzacji, wykraczającej poza standardowe atrybuty kontrolera, wynika z konieczności uwzględnienia dynamicznych reguł dostępu opartych na stanie encji lub specyficznym kontekście. Przykładowo, klient wypożyczalni ma prawo wyłącznie do przeglądania własnego profilu użytkownika, a nie profili innych osób. Weryfikowane jest to poprzez porównanie identyfikatora żądanego rekordu z wartością `_userContext.UserId`. Analogicznie, pracownik biurowy (`OFFICE`) uzyskuje dostęp wyłącznie do pojazdów przypisanych do jego działu, co jest realizowane poprzez powiązanie typu konta z identyfikatorem oddziału, pobieranym za pomocą usługi `IDepartmentAccessService`. Takie reguły nie mogą być efektywnie wyrażone w postaci statycznych atrybutów na poziomie metod kontrolera, dlatego zostały przeniesione do kodu handlera, gdzie towarzyszą walidacji parametrów.

Izolacja kontekstu tenanta jest realizowana jako kaskada przechodząca przez kolejne warstwy systemu. Middleware w warstwie prezentacji identyfikuje tenanta na podstawie subdomeny (np. „xyz” z hosta xyz.wellhicle.com) i przekazuje jego identyfikator do `HttpContext`. Następnie jest on udostępniany handlerom poprzez interfejsy `IUserContext` oraz `ITenantProvider`. W handlerze `ITenantValidationService` dodatkowo sprawdzana jest istnienie i aktywność tenanta. Na poziomie persystencji izolacja danych jest egzekwowana przez jawne filtrowanie po `TenantId` w każdej operacji odczytu lub zapisu (szczegóły w podpunktach *7.1.4.1* i *7.1.4.3*). Repozytoria blokują pobieranie rekordów spoza bieżącego tenanta. Każda z tych warstw może odrzucić żądanie z innego powodu, tworząc wielopoziomowy mechanizm kontroli dostępu – próba uzyskania danych spoza własnego tenanta musiałaby przejść kilka niezależnych etapów walidacji.

Domena dokładająca własne reguły obrony jest niezależna od warstwy infrastruktury. Agregat `User` udostępnia operacje dezaktywacji, aktywacji oraz anonimizacji konta. Ostatnia z nich usuwa dane osobowe i zmienia adres e-mail na losowy, co umożliwia realizację żądań dotyczących RODO (usunięcia danych). Agregat `Reservation` określa uprawnienia do modyfikacji rezerwacji w konkretnym stanie – szczegóły zawarte są w podpunkcie *7.1.2.4*. Te zasady obowiązują niezależnie od tego, czy żądanie pochodzi z API, czy jest wykonywane jako proces tła; dzięki temu semantyka autoryzacji pozostaje spójna w systemie.

7.1.6. Obsługa błędów

Obsługa błędów w projekcie opiera się na rozróżnieniu dwóch kanałów komunikacyjnych: błędów oczekiwanych, które są integralną częścią normalnego przebiegu działania systemu (np. walidacja danych wejściowych, brak zasobu, odmowa dostępu lub niezgodność stanu encji), oraz rzeczywistych awarii infrastrukturalnych (takich jak utrata połączenia z bazą danych, błędy operacji wejścia/wyjścia czy niewłaściwa konfiguracja). Błędy oczekiwane nie są obsługiwane za pomocą wyjątków, lecz propagowane jako wartość zwrotna `Result.Failure` z odpowiednim kodem błędu pochodzącym z enumeracji `ErrorCode` (szczegółowo opisanej w podpunkcie *7.1.3.3*). Rzeczywiste awarie ujawniają się natomiast jako wyjątki i są obsługiwane lokalnie w handlerach lub za pomocą domyślnego mechanizmu ASP.NET Core. Granica między tymi dwoma kanałami jest konsekwentnie zachowana we wszystkich warstwach aplikacyjnych: handler komendy lub zapytania nie generuje wyjątku biznesowego, lecz zwraca wynik. Wyjątek pojawia się jedynie w przypadku odrzucenia argumentu przez konstruktor encji domenowej (np. `Reservation` za pomocą `ArgumentException`) lub gdy operacja zapisu do bazy zakończy się błędem infrastrukturalnym. W obu tych sytuacjach wyjątek jest przechwytywany w bloku try/catch, konwertowany na `Result.Failure` z kodem `VALIDATION_FAILED` lub `GENERIC_ERROR`, a otwarte transakcje

są cofane przy użyciu metody `RollbackTransactionAsync` (przykład przedstawiono w podpunkcie *7.1.3.2*).

Konwersję typu `Result<T>` na odpowiedź HTTP realizuje klasa pomocnicza `ResultExtensions` z pakietu `Wellhicle.Presentation/Api/Helpers`. Klasa ta udostępnia metody rozszerzające dla bazy `ControllerBase`, takie jak `ToOkResult`, `ToCreatedResult`, `ToNoContentResult` oraz dwie wersje `ToActionResult`. Każda z tych metod rozróżnia przypadek sukcesu od wystąpienia błędu. W przypadku sukcesu zwracane są kody odpowiedzi HTTP: 200 OK, 201 Created lub 204 No Content wraz z wartością wyniku. Jeśli wystąpi błąd, metoda wykorzystuje wewnętrzną funkcję konwertującą (*Listing 14*), która generuje ujednolicony obiekt błędu. Obiekt ten – `ErrorDto` z polami `Code` i `Message` – jest jedynym dopuszczalnym formatem odpowiedzi błędnej w całej warstwie API, niezależnie od pochodzenia błędu.

Listing 14 Metoda `CreateErrorResult` z klasy `ResultExtension`, z projektu `Wellhicle.Presentation`. Źródło: opracowanie własne.

```
private static ActionResult CreateErrorResult(ControllerBase controller,
string? errorCode, string? errorMessage)
{
    var errorCodeInt = int.TryParse(errorCode, out var parsedCode)
        ? parsedCode
        : (int)ErrorCode.UNKNOWN_ERROR;
    var originalMessage = string.IsNullOrWhiteSpace(errorMessage)
        ? ErrorMessages.GetMessage(errorCodeInt)
        : errorMessage;

    var isCensored = ErrorMessages.IsCensored(errorCodeInt);

    var publicCode = isCensored
        ? (int)ErrorMessages.GetCensoredErrorCode()
        : errorCodeInt;

    var publicMessage = isCensored
        ? ErrorMessages.GetCensoredMessage()
        : originalMessage;

    LogResolvedError(controller, errorCodeInt, originalMessage,
isCensored);

    return new ObjectResult(new ErrorDto
    {
        Code = publicCode,
        Message = publicMessage
    })
    {
        StatusCode = ErrorMessages.GetHttpStatusCode(errorCodeInt)
    };
}
```

Mechanizm realizuje cztery kroki w ustalonej kolejności. W pierwszym etapie rozpakowuje kod błędu z obiektu `Result` – jeśli nie można go sparsować, traktuje sytuację jako `UNKNOWN_ERROR`. Następnie ustala komunikat lokalny przekazany przez handler lub domyślny ze słownika `ErrorMessages.GetMessage`. W kolejnym kroku sprawdza za pomocą metody `IsCensored`, czy kod wymaga ocenzurowania. Jeżeli tak, zastępuje oryginalne wartości publicznymi odpowiednikami, które nie ujawniają informacji wrażliwych (przykład zastosowania dla logowania w podpunkcie *7.1.5.1*, a proces cenzury w podpunkcie *7.1.3.3*). Następnie wywołuje metodę `LogResolvedError`, która zapisuje do logu oryginalny (nieocenzurowany) kod błędu i komunikat,

dodając odpowiednie tagi do aktualnego wpisu aktywności z wykorzystaniem OpenTelemetry. Dzięki temu system administracyjny widzi rzeczywistą przyczynę błędu, podczas gdy konsument API otrzymuje jedynie ocenzurowaną wersję. Ostatni etap polega na skomponowaniu obiektu `ObjectResult` z danymi `ErrorDto` w ciele odpowiedzi oraz statusem HTTP, pobranym z metody `ErrorMessages.GetHttpStatusCode`, która mapuje kod biznesowy na standardowe kody HTTP (401, 403, 409, 413; dla pozostałych przypadków – 400).

W obecnej implementacji nieobsłużone wyjątki kończą się domyślnym błędem 500 Internal Server Error w ASP.NET Core. Jest to świadomy kompromis: każda operacja biznesowa jest już lokalnie obsługiwana przez wzorzec `Result`, a krytyczne ścieżki zawierają dedykowane bloki try/catch; globalny handler zwracający `ErrorDto` został odłożony jako obszar do dalszego rozwoju. Walidacja danych wejściowych na poziomie warstwy prezentacji jest realizowana przez filtr akcji `AsyncFluentValidationActionFilter`, który uruchamia walidatory FluentValidation przed przekazaniem żądania do handlera. Filtr nie wykorzystuje mechanizmu `Result<T>`, lecz w przypadku niepowodzenia walidacji buduje `ObjectResult` z `ErrorDto` bezpośrednio, korzystając jednak z tej samej infrastruktury (`ErrorMessages.GetMessage`, `IsCensored`, `GetHttpStatusCode`) co `ResultExtensions`. Kod błędu jest pobierany z definicji walidatora, dzięki czemu walidacja schematu DTO oraz reguł biznesowych w handlerach generuje spójne odpowiedzi dla konsumentów API pod względem formatu.

Z perspektywy konsumenta API mechanizm ten oferuje kilka kluczowych właściwości. Po pierwsze, zapewnia ujednolicony kontrakt – każda odpowiedź błędna ma identyczną strukturę { `Code`, `Message` }, niezależnie od warstwy systemu, która wygenerowała błąd. Po drugie, gwarantuje bezpieczeństwo: wrażliwe dane (np. logowanie, autoryzacja konta czy awarie infrastruktury) są maskowane przed ekspozycją na zewnątrz, ale pełna informacja o błędzie trafia do logów, co umożliwia dokładną diagnostykę bez narażania użytkownika. Po trzecie, zapewnia obserwowalność: każdy wystąpiony błąd jest rejestrowany wraz z oryginalnym kodem błędu, komunikatem oraz informacją o cenzurze danych. Dodatkowo mechanizm ten aktywnie integruje się z OpenTelemetry, co ułatwia monitorowanie i analizę zdarzeń. Po czwarte, oferuje przewidywalność po stronie kontrolera – autor punktu końcowego nie musi ręcznie obsługiwać błędów w trybie try/catch, lecz wywołuje funkcje takie jak `result.ToOkResult()` lub `result.ToNoContentResult()`, a mechanizm samodzielnie decyduje, czy odpowiedź powinna być uznana za sukces, czy konwersję błędu.

7.1.7. Obserwowalność

Obserwowalność w projekcie opiera się na dwóch filarach. Pierwszy stanowi logowanie strukturalne realizowane przez Serilog, które konfigurowane jest poprzez odczyt sekcji z pliku appsettings. Konfiguracja definiuje minimalne poziomy logowania, uwzględniając osobne nadpisania dla przestrzeni nazw `Microsoft`, `Microsoft.EntityFrameworkCore` oraz `System`. Każdy wpis jest wzbogacany o właściwości `Application` i `Environment`, a dane są przesyłane do trzech źródeł: konsoli (zawsze), Seq (opcjonalnie) oraz pliku (opcjonalnie). Wszystkie handlery, procesory i usługi otrzymują przez konstruktor instancję `ILogger<T>` i zapisują logi w formacie strukturalnym [43]. Drugi filar to OpenTelemetry, które wykorzystuje trzy rodzaje sygnałów: trace'y, metryki oraz logi. Automatyczna instrumentacja jest włączona dla ASP.NET Core (z obsługą `RecordException = true` i dodatkowymi atrybutami takimi jak `http.request.body`, `http.response.body`, `tenant.id`, `exception.*`), klienta HTTP oraz Entity Framework Core (z atrybutem `db.command.text`). Sygnały są eksportowane do konsoli w trybie deweloperskim, natomiast w środowiskach staging i produkcyjnym trafiają do kolektora OTLP w klastrze Kubernetes [117].

Kluczowym elementem łączącym logi z trace'ami jest `CorrelationIdMiddleware` (*Listing 15*) działające w warstwie prezentacji. Dla każdego żądania odczytuje on identyfikator korelacji (`X-Correlation-Id`), generując nowy GUID, jeśli klient nie przesłał go w nagłówku. Identyfikator jest zapisywany jednocześnie w trzech miejscach: w `HttpContext.Items` (dostępne dla późniejszych komponentów), jako tagi `correlation.id`, `trace.id`, `span.id` na bieżącym Activity (widoczne w OTLP) oraz w `LogContext` Serilog, gdzie każdy log związany z żądaniem zawiera te identyfikatory. Identyfikator jest również zwracany w nagłówku odpowiedzi, co ułatwia debugowanie po stronie klienta API. Dzięki temu rozwiązaniu możliwe jest pełne odtworzenie jednego żądania HTTP: ścieżki przez warstwy (traces), zdarzeń (logs) oraz wyniku (`Result`/HTTP), przy użyciu pojedynczego identyfikatora.

Listing 15 Fragment klasy `CorrelationIdMiddleware` z projektu `Wellhicle.Presentation`. Źródło: opracowanie własne.

```
context.Items[CorrelationIdKey] = correlationId;
context.Response.Headers[CorrelationIdHeader] = correlationId;

Activity.Current?.SetTag("correlation.id", correlationId);
Activity.Current?.SetTag("trace.id",
Activity.Current.TraceId.ToString());
Activity.Current?.SetTag("span.id", Activity.Current.SpanId.ToString());

using (LogContext.PushProperty("CorrelationId", correlationId))
using (LogContext.PushProperty("TraceId",
Activity.Current?.TraceId.ToString()))
using (LogContext.PushProperty("SpanId",
Activity.Current?.SpanId.ToString()))
{
    await _next(context);
}
```

Korelacja nie ogranicza się wyłącznie do warstwy prezentacji. W projekcie `EmailWorker` identyfikator korelacyjny stanowi integralną część obiektu `EmailMessage`, który jest przekazywany przez RabbitMQ. `DeadLetterHandler` rejestruje ten identyfikator podczas przenoszenia wiadomości do DLQ, co umożliwia powiązanie nieudanych prób wysłania e-maila z oryginalnym żądaniem HTTP. W przypadku błędów rozszerzenie `ResultExtensions` (szczegóły opisane w punkcie *7.1.6*) oznacza aktywny tag Activity atrybutami takimi jak `error.code`, `error.message`, `error.censored` oraz `error.name`. Te informacje są następnie przekazywane do tego samego trace'u, w którym znajduje się obsługiwane żądanie. Cały łańcuch operacji – od kontrolera, przez handler, repozytorium i transakcję, aż po asynchroniczną wysyłkę e-maila – jest spójnie powiązany za pomocą jednego identyfikatora korelacyjnego.

7.1.8. Procesy w tle

Procesy działające poza głównym cyklem żądanie-odpowiedź w projekcie Wellhicle można podzielić na dwie kategorie. Pierwsza z nich obejmuje zadania uruchamiane cyklicznie według ustalonego harmonogramu, które są orkiestrowane przez bibliotekę Hangfire i implementowane jako procesory w warstwie aplikacyjnej. Druga kategoria dotyczy przetwarzania przychodzących wiadomości z kolejki RabbitMQ, realizowanego przez odrębny projekt `Wellhicle.EmailWorker` wystawiający usługę typu worker. Granica między tymi dwiema kategoriami przebiega na poziomie sposobu inicjacji pracy: w pierwszym przypadku jest to zegar, natomiast w drugim przychodząca wiadomość.

7.1.8.1. Procesy cykliczne uruchamiane przez Hangfire

Hangfire pełni w projekcie rolę schedulera dla zadań powtarzalnych, uruchamianych w określonych odstępach czasu [118]. Kluczowym elementem rejestracji harmonogramów jest klasa `RecurringJobsRegistration` z pakietu `Wellhicle.Presentation/Startup`, której metoda `Register` wywoływana jest przy starcie aplikacji, przedstawiona w *Listing 16*. Wewnątrz tej metody znajduje się dziesięć wywołań `RecurringJob.AddOrUpdate`. Każde z nich wiąże identyfikator zadania (np. "reservation-no-show") z odpowiednim interfejsem procesora z warstwy aplikacyjnej (np. `INoShowReservationProcessor`) oraz definiuje wyrażenie cron, określające częstotliwość uruchamiania zadania. Interfejs jest następnie rozwiązywany przez Hangfire poprzez kontener wstrzykiwania zależności podczas każdego cyklu, co pozwala utrzymać procesor jako zwykłą klasę warstwy aplikacyjnej, bez bezpośredniej zależności od biblioteki schedulera.

Listing 16 Fragment klasy `RecurringJobsRegistration` z projektu `Wellhicle.Presentation`. Źródło: opracowanie własne.

```
public static class RecurringJobsRegistration {
  public static void Register() {
   RecurringJob.AddOrUpdate<INoShowReservationProcessor>( "reservation-
no-show", p
    => p.ProcessAsync(100, null, CancellationToken.None), "*/10 * * *
*");
   RecurringJob.AddOrUpdate<IExpiredPendingPaymentProcessor>(
        "expired-pending-payments",
        p => p.ProcessAsync(100, null, CancellationToken.None),
        "*/5 * * * *");

    RecurringJob.AddOrUpdate<ISubscriptionBillingGenerationProcessor>(
        "subscription-billing-generation",
        p => p.ProcessAsync(100, null, CancellationToken.None),
        "0 */2 * * *");

    RecurringJob.AddOrUpdate<IOfferLifecycleProcessor>(
        "offer-lifecycle",
        p => p.ProcessAsync(100, null, CancellationToken.None),
        "0 1 * * *");

    // ... pozostałe sześć rejestracji w tej samej konwencji
 }
}
```

Zarejestrowane procesory obsługują kluczowe obszary biznesowe systemu Wellhicle, wykonując zautomatyzowane zadania w określonych interwałach czasowych. W module rezerwacji dwa procesy działają co 10 minut: `NoShowReservationProcessor` oznacza nieodbyte rezerwacje po upływie karencji, a `ReservationReminderProcessor` wysyła przypomnienia przed wynajmem. W zakresie płatności i fakturowania procesor `ExpiredPendingPaymentProcessor` (uruchamiany co 5 minut) unieważnia nieopłacone transakcje, podczas gdy `OverdueUnpaidBillingProcessor` (co 10 minut) oznacza przeterminowane dokumenty rozliczeniowe. Subskrypcjami zarządzają trzy procesory działające rzadziej: generują faktury, obsługują zmiany stanów oraz automatyczne odnowienia (częstotliwość 2 lub 4 godziny). Pozostałe procesy (`OfferLifecycleProcessor`, `InactiveTenantProcessor`, `CloudflareDomainRefreshProcessor`) wykonują zadania w cyklu dziennym lub półgodzinnym.

Wszystkie procesory korzystają ze wspólnego kontraktu IProcessor, zawierającego metodę `ProcessAsync(int limit, DateTime? now, CancellationToken)`, zwracającą `Result<TResultDto>`. W większości przypadków jest to `ProcessorResultDto` (z polami:

`Scanned`, `Processed`, `Failed`), a w specjalistycznych wariantach – dedykowane DTO (np. `ExpiredPendingPaymentsResultDto` dla płatności). Parametr limit ogranicza liczbę przetwarzanych elementów na cykl, zaś opcjonalny now umożliwia testowe sterowanie czasem wykonania. Każdy procesor stosuje identyczny wzorzec: pobiera dane z repozytorium (stronicowanie), iteruje po wynikach i przetwarza każdy element w osobnej transakcji (try/begin/commit/catch/rollback – szczegóły opisano w sekcji *7.1.4.1*). Awaria jednego procesu nie wpływa na pozostałe: błędy są logowane, licznik `Failed` aktualizowany, a przetwarzanie kontynuowane.

7.1.8.2. EmailWorker jako konsumer kolejki

`Wellhicle.EmailWorker` to osobny projekt .NET w formie usługi pracującej (ang. *worker service* [119]), uruchamiany jako niezależny proces obok głównej aplikacji. Decyzja o wydzieleniu tego komponentu wynika z trzech kluczowych przesłanek: izolacji awarii, co pozwala na utrzymanie stabilności działania systemu w przypadku problemów z serwerem SMTP; niezależnego skalowania, umożliwiającego łatwe dostosowywanie liczby wystąpień usługi do rosnących obciążeń (np. większej liczby wysyłanych wiadomości); oraz oddzielnego cyklu wdrożeniowego, co oznacza, że zmiany wprowadzone w komponencie pocztowym nie wymagają przebudowy całej aplikacji. Mimo pewnej współdzielonej infrastruktury – takiej jak konfiguracja logowania Serilog, wybrane abstrakcje z `Wellhicle.Application.Abstractions` (np. `IClock`) oraz model danych poprzez referencję do `Wellhicle.Infrastructure` – usługa `EmailWorker` działa w runtime jako odrębny host.

Przepływ wiadomości umożliwia komunikację między stronami systemu. Aplikacja główna publikuje wiadomość za pomocą metody `IDomainEventPublisher.PublishEmailMessageAsync`, zaimplementowanej przez komponent `RabbitMqDomainEventPublisher`, który serializuje obiekt i umieszcza go w kolejce RabbitMQ. Po stronie odbiorcy klasa `EmailConsumerService`, dziedzicząca po `BackgroundService` z pakietu `Microsoft.Extensions.Hosting`, nasłuchuje na kolejce i dla każdej odebranej wiadomości wykonuje ustaloną sekwencję operacji: pobiera odpowiedni szablon e-maila, waliduje zmienne, komponuje nagłówek oraz treść wiadomości, wybiera konto SMTP (należące do tenanta lub systemowe) i wysyła wiadomość przy użyciu obiektu SmtpClientFactory.

Niezawodność dostarczania wiadomości jest wspierana przez szereg mechanizmów. Po stronie RabbitMQ automatyczne odzyskiwanie (`AutomaticRecoveryEnabled = true`, `NetworkRecoveryInterval = 10s`) eliminuje konieczność restartu workera w przypadku chwilowej utraty połączenia z brokerem. W warstwie SMTP stosowany jest obwód bezpieczeństwa oparty na bibliotece Polly: gdy ponad 50% prób dostarczenia wiadomości kończy się wyjątkiem `SmtpException` lub `TimeoutException` w określonym przedziale czasowym, proces zostaje przerwany i wznawiany dopiero po skonfigurowanym okresie odpoczynku. Wiadomości niedostarczone po ponowieniach są przekazywane do kolejki dead-letter przez `DeadLetterHandler`, a `EmailRetryHandler` koordynuje politykę retries. Stan połączenia z brokerem jest dodatkowo monitorowany za pomocą RabbitMQ Health Check. Proces zatrzymania workera obsługiwany jest w sposób gładki (graceful shutdown). EmailConsumerService śledzi liczbę aktywnych zadań przetwarzania i czeka maksymalnie 30 sekund na ich ukończenie przed zwolnieniem połączenia.

Dzięki takiemu podejściu system osiąga wyraźne rozdzielenie ról: aplikacja główna nie ma wiedzy na temat czasu fizycznego dostarczenia wiadomości, trwania procesu ani ewentualnej niedostępności serwera SMTP. Z perspektywy aplikacji publikacja do kolejki jest operacją synchroniczną zakończoną natychmiastowo. Cała logika rzeczywistego dostarczenia, włącznie z polityką ponawiania prób, izolacją awarii oraz monitorowaniem, jest realizowana przez serwis roboczy.

7.1.9. Przechowywanie plików biznesowych

Mechanizm przechowywania plików biznesowych w serwerze aplikacyjnym Wellhicle obejmuje zasoby takie jak zdjęcia pojazdów, pliki strony wypożyczalni, szablony dokumentów oraz rozliczenia. Rozwiązanie nie zapisuje binarnej zawartości bezpośrednio w bazie danych – zamiast tego oddziela metadane od fizycznej treści: rekordy domenowe są utrzymywane w tabeli `Files`, natomiast same pliki przechowywane są w magazynie RustFS kompatybilnym z protokołem S3.

Taki podział jest spójny z architekturą *backendu* opartą na separacji odpowiedzialności: baza danych gromadzi informacje niezbędne dla logiki domenowej, takie jak typ pliku, nazwa wyświetlana, typ `MIME`, rozmiar czy powiązanie z tenantem lub encją. Magazyn S3/RustFS odpowiada natomiast za trwałe przechowywanie danych binarnych – model domenowy pozostaje wolny od szczegółów infrastruktury, a baza nie pełni roli repozytorium dużych obiektów.

7.1.9.1. Architektura i podział odpowiedzialności

Obsługa plików jest zorganizowana wokół dwóch poziomów abstrakcji. `IDocumentService` pełni rolę biznesowego serwisu wykorzystywanego w logice aplikacyjnej związanej z dokumentami i plikami. Przez niego przechodzi każdy przepływ zapisu, pobrania adresu lub usunięcia pliku. Pod spodem serwis korzysta z `IFileStorageService`, czyli technicznego kontraktu do bezpośrednich operacji na magazynie S3, takich jak zapis strumienia, usunięcie obiektu i wygenerowanie adresu dostępu. Przykładowe metody interfejsu `IDocumentService` zostały przedstawione na *Listing 17* Dzięki temu każdy plik może być obsługiwany w spójnym przepływie aplikacyjnym, a szczegóły klienta S3, zbioru (ang. *bucket)* i sposobu podpisywania adresów pozostają w warstwie infrastruktury.

Listing 17 Wybrane metody z interfejsu IDocumentService. Źródło: opracowanie własne.

```
public interface IDocumentService
{
    Task<Result<(string Path, string Url)>> UploadFile<T>(
        int? tenantId,
        string fileName,
        string contentType,
        Stream stream,
        CancellationToken cancellationToken,
        int? entityId) where T : File;

    Task<Result> DeleteFile<T>(
        string filename,
        CancellationToken cancellationToken) where T : File;

    string GetFileUrl<T>(string filename) where T : File;
}
```

7.1.9.2. Organizacja magazynu i typy plików

Organizacja magazynu opiera się na dwóch logicznych zbiorach: publicznym i prywatnym. Do zbioru publicznego trafiają pliki, które mogą być udostępniane bez indywidualnego podpisywania adresu, takie jak zdjęcia pojazdów lub regulamin wypożyczalni. Zbiór prywatny przechowuje pliki wymagające kontroli dostępu, np. dokumenty rozliczeniowe, szablony czy podpisy pracowników powiązane z wynajmem.

Dodatkowo konfiguracja mapuje typ encji plikowej na folder (prefiks obiektu w zbiorze). Zdjęcia pojazdów, dokumenty strony, szablony i dokumenty rozliczeniowe trafiają do odrębnych prefiksów. Decyzja o przypisaniu pliku do zbioru publicznego lub prywatnego jest określana w konfiguracji połączenia z magazynem, przedstawionej na *Listing 18*. Oznacza to, że dodanie nowej kategorii plików wymaga utworzenia odpowiedniej klasy domenowej oraz uwzględnienia jej w konfiguracji folderów i zbiorów.

Listing 18 Fragment pliku appsettings.json przedstawiający konfigurację zbiorów i folderów w magazynie RustFS. Źródło: opracowanie własne.

```
"Buckets": {
  "Public": "public",
  "Private": "private"
},
"Folders": {
  "VehiclePhoto": {
    "Folder": "vehicles",
    "IsPublic": true
  },
  "VehicleGroupPhoto": {
    "Folder": "vehicle-groups",
    "IsPublic": true
  },
  "ServicePhoto": {
    "Folder": "services",
    "IsPublic": false
  }
}
```

7.1.9.3. Klucze obiektów i adresy dostępu

Klucz obiektu w magazynie jest budowany z kilku elementów: identyfikatora tenanta, prefiksu odpowiadającego domenowemu typowi pliku, opcjonalnego identyfikatora encji powiązanej oraz losowej nazwy pliku zachowującej rozszerzenie. Oryginalna nazwa przesłana przez użytkownika nie jest wykorzystywana jako nazwa obiektu w magazynie – trafia ona do pola `DisplayFileName`, natomiast pole `FileName` przechowuje faktyczny klucz obiektu.

Logika uzyskiwania URL do obiektu została przedstawiona na *Listing 19*. W przypadku plików prywatnych właściwy adres pobrania jest generowany na żądanie jako presigned URL, czyli czasowy link dostępowy. Dla plików publicznych możliwe jest utworzenie zwykłego adresu prowadzącego do obiektu w publicznym zbiorze.

Listing 19 Fragment klasy RustFileStorageService odpowiedzialny za generowanie adresów URL obiektów. Źródło: opracowanie własne.

```
public class RustFileStorageService : IFileStorageService{

    public string GetDownloadUrl<T>(string filename) where T : Entity
    {
        //zwraca zbiór i folder dla typu encji
        var folderConf = _getFolderAndBucketForEntity<T>();
        //zwraca rzeczywistą nazwę zbioru w magazynie
        var bucket = _resolveBucket(folderConf.bucket);

        if (folderConf.bucket == StorageBucket.Private)
        {

            return _getPresignedUrl<T>(filename, bucket);
        }
        else
        {
            return string.Concat(
                _settings.ServiceURL, "/", bucket, "/", filename);
        }
    }

    private string _getPresignedUrl<T>(
    string filename,
    string bucketName) where T : Entity
    {
     var presignRequest = new GetPreSignedUrlRequest
     {
         BucketName = bucketName,
         Key = filename,
         Protocol = _settings.UseHttp ?
            Protocol.HTTP : Protocol.HTTPS,
         Verb = HttpVerb.GET,
         Expires = _clock.UtcNow.AddMinutes(
            _settings.PresignUrlExpirationMinutes)
     };

     var url = _s3.GetPreSignedURL(presignRequest);
     return url;
    }
}
```

7.1.9.4. Domenowe modelowanie plików

Na poziomie domeny pliki są reprezentowane przez hierarchię typów odpowiadających ich roli biznesowej, zamiast przez jeden ogólny rekord rozróżniany jedynie kategorią. Wspólna klasa bazowa *File* zawiera pola wspólne dla wszystkich plików, natomiast klasy pochodne opisują konkretne przypadki biznesowe:

- `BillingDocument` – dokument rozliczeniowy,
- `RentalDocument` – plik związany rezerwacją (np. umowa lub protokół zdawczo-odbiorczy),
- `Template` – szablon do generowania `BillingDocument` i `RentalDocument`
- `GlobalTemplate` – domyślny szablon do generowania `BillingDocument` i `RentalDocument`, jeśli nie zostały wgrane szablony przez właściciela,

- `SiteDocument` – dokument publiczny strony wypożyczalni (np. regulamin czy polityka prywatności),
- `VehiclePhoto` – zdjęcie pojazdu,
- `VehicleConditionPhoto` – zdjęcie stanu technicznego pojazdu,
- `ServicePhoto` – zdjęcie z serwisu pojazdu,
- `EmployeeSignature` – podpis pracownika zapisany jako plik PNG (osadzany na dokumentach).

Dziedziczenie umożliwia utrzymanie wspólnego mechanizmu przechowywania przy jednoczesnym powiązaniu typu pliku z konkretną semantyką biznesową (np. rozliczeniem, szablonem czy stanem pojazdu). Dzięki temu kod może korzystać z uniwersalnych operacji magazynowych, ale działa na typach odpowiadających rzeczywistym potrzebom biznesowym.

Od strony bazy danych ta hierarchia jest realizowana w jednej tabeli `Files` z wykorzystaniem strategii TPH (*table-per-hierarchy*). Wspólne pola są przechowywane w jednym miejscu, a typ rekordu identyfikuje dyskryminator `FileType`. Przykład konfiguracji TPH przedstawiono na *Listing 20*.

Listing 20 Fragment konfiguracji zapisu do tabeli Files wykorzystujący TPH. Źródło: opracowanie własne.

```
public class FileConfiguration : IEntityTypeConfiguration<DomainFile>
{
    public void Configure(EntityTypeBuilder<DomainFile> builder)
    {
        builder.ToTable("Files");

        builder.Property(f => f.FileType)
            .IsRequired()
            .HasConversion<int>();

        // Konfiguracja TPH (Table Per Hierarchy) dla dziedziczenia
        builder.HasDiscriminator(f => f.FileType)
            .HasValue<SiteDocument>(FileType.SITE_DOCUMENT)
            .HasValue<EmployeeSignature>(FileType.SIGNATURE)
            .HasValue<Template>(FileType.TEMPLATE)
            .HasValue<RentalDocument>(FileType.RENTAL_DOCUMENT)
            .HasValue<VehiclePhoto>(FileType.VEHICLE_PHOTO)
            .HasValue<VehicleConditionPhoto>(
                FileType.VEHICLE_CONDITION_PHOTO)
            .HasValue<ServicePhoto>(FileType.SERVICE_PHOTO)
            .HasValue<VehicleGroupPhoto>(FileType.VEHICLE_GROUP_PHOTO)
            .HasValue<BillingDocument>(FileType.BILLING)
            .HasValue<GlobalTemplate>(FileType.GLOBAL_TEMPLATE);
    }
}
```

7.1.9.5. Walidacja, zapis plików i metadanych

Przepływ zapisu pliku rozpoczyna się od walidacji wykonywanej na granicy API. Backend sprawdza, czy plik został przesłany, czy nie jest pusty, czy mieści się w dopuszczalnym limicie rozmiaru oraz czy jego nazwa nie zawiera konstrukcji niebezpiecznych, takich jak sekwencje path traversal, separatory ścieżek, znaki sterujące lub bajt zerowy. Oprócz deklarowanego typu MIME weryfikowana jest również sygnatura binarna zawartości, co ogranicza ryzyko zapisania pliku podszywającego się pod inny format. Logika walidacji została przedstawiona na *Listing 21* natomiast jej reguły na *Listing 22*.

Listing 21 Metoda sprawdzająca poprawność przesłanego pliku. Źródło: opracowanie własne.

```
public async Task<SecureFileValidationResult> ValidateAsync(
```

```
    IFormFile file,
    AttachmentRule rule,
    CancellationToken cancellationToken = default)
{
    if (file == null)
        return Invalid("File is required.");

    if (file.Length <= 0)
        return Invalid("File is empty.");

    if (!IsFileNameSafe(file.FileName))
        return Invalid("Invalid file name. Path traversal, null bytes or
invalid characters are not allowed.");

    if (file.Length > rule.EffectiveMaxSizeBytes)
        return Invalid(
            ErrorCode.FILE_TOO_LARGE,
            $"File size exceeds maximum allowed
({rule.EffectiveMaxSizeBytes / (1024 * 1024)} MB).");

    await using var stream = file.OpenReadStream();
    var detectedFileType = await
_streamProvider.TryFindUnambiguousAsync(stream, cancellationToken);

    if (detectedFileType == null)
        return Invalid("File type could not be determined. The file may
be corrupted or of an unsupported type.");

    var detectedMime = detectedFileType.MimeTypes.FirstOrDefault();
    if (string.IsNullOrEmpty(detectedMime))
        return Invalid("Detected file type has no MIME type.");

    if (!rule.MimeTypes.Contains(detectedMime,
StringComparer.OrdinalIgnoreCase))
        return Invalid(
            ErrorCode.INVALID_FILE_TYPE,
            $"File type '{detectedMime}' is not allowed. Allowed types:
{string.Join(", ", rule.MimeTypes)}.",
            detectedMime);

    return new SecureFileValidationResult(true, detectedMime, null,
null);
}
```

Listing 22 Reguły walidacji plików. Źródło: opracowanie własne.

```
public static class AttachmentRules
{
    //Domyślny maksymalny rozmiar pliku (10 MB), gdy typ nie ma własnego
limitu.
    public const long DefaultMaxFileSizeBytes = 10 * 1024 * 1024;

    public static readonly AttachmentRule ImagePhoto = new(
        ["image/jpeg", "image/png"],
        MaxSizeBytes: 10 * 1024 * 1024);

    public static readonly AttachmentRule Docx = new(
        ["application/vnd.openxmlformats-
officedocument.wordprocessingml.document"],
        MaxSizeBytes: null);

    public static readonly AttachmentRule Pdf = new(
        ["application/pdf"],
        MaxSizeBytes: null);

    public static readonly AttachmentRule Svg = new(
        ["image/svg+xml"],
        MaxSizeBytes: 2 * 1024 * 1024);

    public static readonly AttachmentRule Ico = new(
        ["image/x-icon"],
        MaxSizeBytes: 64 * 1024);

    public static readonly AttachmentRule Signature = new(
        ["image/png"],
        MaxSizeBytes: 512 * 1024);
}
```

Po pozytywnej walidacji zawartość jest przesyłana do magazynu, ponieważ dopiero wtedy znany jest końcowy klucz obiektu. Następnie tworzona jest encja domenowa z metadanymi i zapisywana w bazie danych. Taka kolejność jest naturalna, ale wprowadza ryzyko powstania osieroconego obiektu – jeżeli przesyłanie pliku (ang. *upload*) zakończy się sukcesem, a zapis metadanych nie powiedzie się. To ryzyko zostało ograniczone dzięki pomocniczemu przepływowi przedstawionemu na *Listing 23*, który w przypadku błędu próbuje usunąć świeżo przesłany plik.

Listing 23 Mechanizm usuwania pliku po nieudanym zapisie metadanych. Źródło: opracowanie własne.

```
public static async Task<Result<TEntity>>
PersistUploadedEntityAsync<TStoredFile, TUploadResult, TEntity>(
     this IDocumentService documentService,
     Result<TUploadResult> uploadResult,
     Func<TUploadResult, string> pathSelector,
     Func<TUploadResult, TEntity> createEntity,
     Func<TEntity, CancellationToken, Task> persistAsync,
     CancellationToken cancellationToken) where TStoredFile : File
 {
     if (uploadResult.IsFailure)
     {
         return uploadResult.ToResult<TEntity, TUploadResult>();
     }

     var uploadedValue = uploadResult.Value!;
     var uploadedPath = pathSelector(uploadedValue);

     TEntity entity;
     try
     {
```

```
            entity = createEntity(uploadedValue);
        }
        catch (ArgumentException ex)
        {
            await TryDeleteUploadedFileAsync<TStoredFile>(documentService,
uploadedPath, cancellationToken);
            return Result<TEntity>.Failure(ex.Message,
ErrorCode.VALIDATION_FAILED);
        }
        catch (InvalidOperationException ex)
        {
            await TryDeleteUploadedFileAsync<TStoredFile>(documentService,
uploadedPath, cancellationToken);
            return Result<TEntity>.Failure(ex.Message,
ErrorCode.INVALID_OPERATION);
        }

        try
        {
            await persistAsync(entity, cancellationToken);
            return Result<TEntity>.Success(entity);
        }
        catch
        {
            await TryDeleteUploadedFileAsync<TStoredFile>(documentService,
uploadedPath, cancellationToken);
            return
Result<TEntity>.Failure(ErrorCode.FILE_PERSISTENCE_FAILED);
        }
    }
```

7.1.9.6. Usuwanie plików

Osobnym zagadnieniem jest usuwanie plików. Backend wykorzystuje w tym celu interceptor Entity Framework Core – *`FileStorageCleanupInterceptor`* usuwa obiekt z magazynu po udanym usunięciu rekordu pliku z bazy danych. Metoda odpowiedzialna za to działanie została przedstawiona na Listing 24.

Kolejność operacji jest kluczowa: najpierw potwierdzany jest zapis zmian w bazie, a dopiero później wykonywane jest kasowanie obiektu binarnego. Dzięki temu system minimalizuje ryzyko utraty pliku w sytuacji, gdy operacja bazodanowa zostanie odrzucona.

Listing 24 Fragment interceptora odpowiedzialny za kasowanie plików z magazynu. Źródło: opracowanie własne.

```
public override async ValueTask<int> SavedChangesAsync(
    SaveChangesCompletedEventData eventData,
    int result,
    CancellationToken cancellationToken = default)
{
    if (_pendingDeletes.Count == 0)
        return await base.SavedChangesAsync(eventData, result,
cancellationToken);
    foreach (var (entityType, fileName) in _pendingDeletes)
    {
        await _fileStorageService.DeleteAsync(entityType, fileName);
    }
    _pendingDeletes.Clear();
```

```
    return await base.SavedChangesAsync(eventData, result,
cancellationToken);
}
public override void SaveChangesFailed(DbContextErrorEventData eventData)
{
    _pendingDeletes.Clear();
    base.SaveChangesFailed(eventData);
}
```

Mechanizm usuwania ma jednak ograniczenia typowe dla integracji bazy danych z zewnętrznym magazynem. Usunięcie rekordu w bazie i usunięcie obiektu w S3 nie stanowią jednej transakcji. Jeśli zapis w bazie zakończy się powodzeniem, a późniejsze usunięcie obiektu ze storage nie powiedzie się, może pozostać osierocony plik.

7.1.10. Generowanie i przetwarzanie dokumentów

Mechanizm generowania dokumentów uzupełnia moduły rezerwacji, wydań i zwrotów pojazdów oraz rozliczeń. W systemie Wellhicle dokument jest tworzony jako wynik połączenia szablonu DOCX, danych domenowych i konwersji do PDF. Rozwiązanie to jest wykorzystywane między innymi przy przygotowywaniu umów najmu, protokołów wydania lub zwrotu pojazdu oraz dokumentów związanych z rozliczeniem.

Dzięki takiemu podejściu dokumenty zachowują układ przygotowany w edytorze tekstu, ale ich treść wynika z aktualnego stanu systemu. *Backend* może więc generować dokument zawierający dane rezerwacji, pojazdu, klienta wypożyczalni, wypożyczalni albo rozliczenia bez ręcznego przepisywania tych informacji przez pracownika.

7.1.10.1. Podział odpowiedzialności

Architektura generatora została podzielona na dwie części. *Backend* .NET odpowiada za wybór szablonu, pobranie go z magazynu, podstawienie danych domenowych i zapis wyniku. Osobny serwis odpowiada tylko za przekształcenie gotowego dokumentu DOCX do PDF.

Listing 25 Fragment metody `GenerateDocument()` odpowiedzialnej za główny przepływ generatora. Źródło: opracowanie własne.

```
public async Task<Result<(string Path, string Url, string ContentType,
int Size, string DisplayName)>> GenerateDocument<T>(
    int tenantId,
    List<MergeFieldDefinition> fields,
    MergeContext context,
    string templateFilename,
    TemplateType templateType,
    CancellationToken cancellationToken,
    bool isGlobal = false) where T : File
{
    Stream fileStream;
    try
    {
        if (isGlobal)
        {
            fileStream = await _fileStorageService
                .DownloadAsync<GlobalTemplate>(templateFilename);
        }
        else
        {
```

```
            fileStream = await _fileStorageService
                .DownloadAsync<Template>(templateFilename);
        }
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Failed to download template:
{templateName}.", templateFilename);
        return Result<(string Path, string Url, string ContentType, int
Size, string DisplayName)>
            .Failure(ErrorCode.FILE_STORAGE_ERROR);
    }
    await using (fileStream)
    {
        try
        {
            using (var memoryStream = new MemoryStream())
            {
                await fileStream.CopyToAsync(memoryStream,
cancellationToken);
                memoryStream.Position = 0;
                using (var wordDoc =
WordprocessingDocument.Open(memoryStream, true))
                {
                    _resolveWordMergeFields(wordDoc, fields, context);
                }
                memoryStream.Position = 0;

                var docBytes = memoryStream.ToArray();
                var pdf = await _pdfGenerator.GeneratePdfAsync(docBytes,
cancellationToken);

                if (pdf == null)
                {
                    _logger.LogError("PDF generation failed for template:
{templateName}.", templateFilename);
                    return Result<(string Path, string Url, string
ContentType, int Size, string DisplayName)>

.Failure(ErrorCode.DOCUMENT_PDF_GENERATION_FAILED);
                }

                var pdfStream = new MemoryStream(pdf);
```

Centralną rolę po stronie *backendu* pełni `DocumentService` z metodą `GenerateDocument()` przedstawioną na *Listing 25*. Serwis pobiera szablon z magazynu, otwiera go w pamięci, modyfikuje pola w DOCX, wywołuje konwersję PDF przez interfejs `IPdfGenerator`, a następnie zapisuje wynik jako plik biznesowy zgodnie z mechanizmami opisanymi w punkcie *7.1.9*. W zależności od procesu dokument może zostać utrwalony jako `RentalDocument` albo `BillingDocument`.

Punktem wejścia generatora jest szablon DOCX zapisany jako Template albo GlobalTemplate. Backend traktuje plik DOCX jako edytowalny pakiet OpenXML, kopiuje jego zawartość do `MemoryStream`, a następnie otwiera go przez metodę `WordprocessingDocument.Open()`. Po zakończeniu zmian zmodyfikowane bajty dokumentu są przekazywane do konwersji.

7.1.10.2. Mapowanie danych domenowych

Mapowanie danych do pól szablonu opiera się na klasach *MergeFieldDefinition* oraz *MergeContext*. Definicja pola określa nazwę występującą w szablonie, obiekt główny, ścieżkę właściwości i typ pola. *RootObject* wskazuje źródło danych – może to być np. *Reservation*, *Billing*, *Tenant*, *Vehicle* lub zakres systemowy. *PropertyPath* opisuje przejście po właściwościach obiektu.

Wartości pól rozwiązuje mechanizm implementowany przez klasę *MergeFieldResolver* z kluczowymi metodami służącymi do ustalenia, jakie wartości mają zostać osadzone na dokumencie:

- `Resolve()` – przedstawiona na Listing 26,
- `ResolvePath()` – przestawiona na Listing 27.

Mechanizm wykorzystuje refleksję: najpierw identyfikuje właściwy obiekt z kontekstu (*MergeContext)*, a następnie przechodzi po kolejnych elementach ścieżki właściwości. Daje to elastyczność przy projektowaniu szablonów, ale ma też ograniczenie: zmiany nazw właściwości w modelu domenowym lub błędna ścieżka mogą skutkować pustą wartością w dokumencie.

Listing 26 Metoda `Resolve()`, służąca do zarządzania przepływem rozwiązywania pól na szablonie. Źródło: opracowanie własne.

```
public MergeFieldResult Resolve(
string mergeKey,
IEnumerable<MergeFieldDefinition> fields,
MergeContext context)
{
    var field = fields.FirstOrDefault(x => x.MergeKey == mergeKey);
    if (field == null)
        return new MergeFieldResult { StandardField = string.Empty };

    if (field.RootObject == "System")
    {
        switch (field.MergeKey)
        {
            case "##DATE##":
                return new MergeFieldResult { StandardField
                    = _clock.UtcNow.ToString("dd-MM-yyyy") };
            default:
                return new MergeFieldResult { StandardField
                    = string.Empty };
        }
    }
    var root = GetRoot(context, field.RootObject);
    if (field.PropertyPath.Contains("+"))
    {
        var sb = new StringBuilder();
        foreach (var value in field.PropertyPath.Split("+"))
        {
            if (sb.Length > 0)
                sb.Append(" ");
            sb.Append(ResolvePath(root, value));
        }
        return new MergeFieldResult { StandardField = sb.ToString()};
    }

    var result = ResolvePath(root, field.PropertyPath,
        field.MergeFieldType == MergeFieldType.MONEY);
    if (result is IEnumerable enumerable && result is not string)
    {
```

```
            return new MergeFieldResult { TableContent
                = enumerable.Cast<object>() };
        }
        return new MergeFieldResult { StandardField = result?.ToString()
?? string.Empty };
    }
```

Listing 27 Metoda `ResolvePath()` odczytująca rzeczywiste wartości z właściwości obiektów. Źródło: opracowanie własne.

```
private static object? ResolvePath(object? source, string path, bool
isMoney = false)
{
    if (source == null)
        return null;

    object? current = source;

    foreach (var segment in path.Split('.'))
    {
        if (current == null)
            return null;

        var cacheKey = $"{current.GetType().Name}.{segment}";
        var prop = _propertyCache.GetOrAdd(cacheKey, key =>
        {
            return current.GetType().GetProperty(segment);
        });

        if (prop == null)
            throw new InvalidOperationException(
                $"Property '{segment}' not found on
'{current.GetType().Name}'");

        current = prop.GetValue(current);
    }
    if (isMoney)
    {
        var money = Convert.ToDouble(current) / 100;
        return money;
    }
    return current;
}
```

7.1.10.3. Pola DOCX i tabele

Generator obsługuje klasyczne pola *MERGEFIELD* oraz pola *SimpleField*. W obu przypadkach kod musi najpierw wydobyć nazwę pola – co zostało przedstawione na *Listing 28*, odszukać mapowanie, rozwiązać wartość z kontekstu i zastąpić tekst wyniku. Nie jest to zwykłe wyszukiwanie tekstu, ponieważ Microsoft Word zapisuje pola jako elementy struktury OpenXML.

Listing 28 Metoda `TryExtractMergeKey()` służąca do znajdowania pól tymczasowych w dokumencie. Źródło: opracowanie własne.

```
private static bool TryExtractMergeKey(string? instruction, out string
mergeKey)
{
    mergeKey = string.Empty;

    if (string.IsNullOrWhiteSpace(instruction))
        return false;
```

```
    var match = Regex.Match(
        instruction,
        @"MERGEFIELD\s+([^\s\\]+)",
        RegexOptions.IgnoreCase);

    if (!match.Success)
        return false;

    mergeKey = match.Groups[1].Value.Trim();
    return true;
}
```

Osobnym przypadkiem są tabele zawierające kolekcje danych. Mechanizm ten ma znaczenie przede wszystkim w dokumentach rozliczeniowych, gdzie trzeba przedstawić listę pozycji. Generator rozpoznaje wiersz tabeli jako wzorzec kolekcji, klonuje go dla każdego elementu listy, podstawia wartości i usuwa pierwotny wiersz-szablon. Mechanizm ten został przedstawiony na *Listing 29*.

Listing 29 Metoda `_expandCollectionTemplateRows()` odpowiedzialna za stworzenie wiersza tabeli dla każdego elementu kolekcji. Źródło: opracowanie własne.

```
private void _expandCollectionTemplateRows(
    DocumentFormat.OpenXml.Wordprocessing.Document document,
    IEnumerable<MergeFieldDefinition> fields,
    MergeContext context)
{
    var fieldMap = fields.ToDictionary(f => f.MergeKey,
StringComparer.Ordinal);
    var rows = document.Descendants<TableRow>().ToList();

    foreach (var row in rows)
    {
        var bindings = _getCollectionBindings(row, fieldMap, context);
        if (bindings.Count == 0)
            continue;

        var firstBinding = bindings[0];
        if (bindings.Any(b => !string.Equals(b.BindingKey,
firstBinding.BindingKey, StringComparison.Ordinal)))
            continue;

        foreach (var item in firstBinding.Items)
        {
            var clonedRow = (TableRow)row.CloneNode(true);

            foreach (var binding in bindings)
            {
                var value = _resolveCollectionBindingValue(binding,
context, item);
                _replaceMergeFieldValue(clonedRow,
binding.Definition.MergeKey, value);
            }

            row.InsertBeforeSelf(clonedRow);
        }

        row.Remove();
    }
}
```

7.1.10.4. Konwersja DOCX na PDF

Po wypełnieniu dokumentu *backend* przekazuje bajty DOCX do `IPdfGenerator`. Implementacja `PdfGenerator` przedstawiona na *Listing 30* jest prostym adapterem HTTP. Tworzy żądanie multipart, dodaje dokument jako plik formularza i wysyła go do punktu końcowego (ang. *endpoint*) `/convert/docx-to-pdf`.

Listing 30 Klasa `PdfGenerator` wraz z metodą `GeneratePdfAsync()`, odpowiedzialną za komunikację z FastAPI. Źródło: opracowanie własne.

```
public class PdfGenerator : IPdfGenerator
{
    private readonly HttpClient _http;

    public PdfGenerator(HttpClient http)
    {
        _http = http;
    }

    public async Task<byte[]> GeneratePdfAsync(
        byte[] docx,
        CancellationToken ct)
    {
        using var content = new MultipartFormDataContent();
        content.Add(
            new ByteArrayContent(docx),
            "file",
            "input.docx");

        using var response = await _http.PostAsync(
            "/convert/docx-to-pdf",
            content,
            ct);

        response.EnsureSuccessStatusCode();

        return await response.Content.ReadAsByteArrayAsync(ct);
    }
}
```

Serwis konwertujący przedstawiony na *Listing 31* został zaimplementowany jako aplikacja FastAPI. Przyjmuje plik DOCX, zapisuje go tymczasowo w folderze */tmp*, uruchamia *libreoffice --headless --convert-to pdf*, a następnie zwraca wygenerowany PDF.

Listing 31 Fragment punktu końcowego FastAPI odpowiedzialnego za konwertowanie DOCX na PDF. Źródło: opracowanie własne.

```
@app.post("/convert/docx-to-pdf")
async def convert(file: UploadFile, background_tasks: BackgroundTasks):

    if not file.filename.endswith(".docx"):
        raise HTTPException(
            status_code=400,
            detail="Only .docx files are supported"
        )

    docx_path = f"/tmp/{uuid4()}.docx"
    pdf_path = docx_path.replace(".docx", ".pdf")

    try:
        with open(docx_path, "wb") as f:
            f.write(await file.read())
```

```
        result = subprocess.run(
                [
                    "libreoffice",
                    "--headless",
                    "--nologo",
                    "--nofirststartwizard",
                    "--convert-to", "pdf",
                    "--outdir", "/tmp",
                    docx_path
                ],
                capture_output=True,
                text=True
            )

        if result.returncode != 0:
            raise HTTPException(
                status_code=500,
                detail={
                    "message": "LibreOffice conversion failed",
                    "stderr": result.stderr
                }
            )

        if not os.path.exists(pdf_path):
            raise HTTPException(
                status_code=500,
                detail="PDF file was not created"
            )

        background_tasks.add_task(cleanup_files, docx_path, pdf_path)

        return FileResponse(
            pdf_path,
            media_type="application/pdf",
            filename="output.pdf"
        )
```

7.1.10.5. Podpisywanie dokumentów

Generator obsługuje również podpisywanie dokumentów najmu. W szablonie mogą znajdować się specjalne znaczniki przeznaczone na podpisy pracownika i klienta wypożyczalni. Podczas wypełniania pliku DOCX zostają one zamienione przez pomocniczy przepływ przedstawiony na *Listing 32* na ukryte znaczniki, które są następnie przenoszone do pliku PDF.

Listing 32 Metoda `_tryHandleSignatureMergeField()` ukrywająca znaczniki podpisów. Źródło: opracowanie własne.

```
private static bool _tryHandleSignatureMergeField(
    string mergeKey,
    IEnumerable<DText> texts)
{
    if (!SignatureMergeFields.Contains(mergeKey))
        return false;

    var textList = texts.ToList();
    if (!textList.Any())
        return true;

    foreach (var text in textList)
        text.Text = string.Empty;
```

```
    textList.First().Text = mergeKey;

    foreach (var run in textList
                 .Select(t => t.Parent as Run)
                 .Where(r => r != null)
                 .Distinct())
    {
        _applyHiddenSignatureMarkerStyle(run!);
    }

    return true;
}
```

Etap podpisywania pobiera gotowy plik PDF oraz obrazy podpisów, lokalizuje odpowiednie znaczniki i osadza je w dokumencie. Jeżeli znacznik nie zostanie znaleziony, stosowane jest awaryjne położenie na ostatniej stronie pliku PDF. Rozwiązanie to umożliwia projektowanie miejsca podpisu w programie Microsoft Word, natomiast faktyczne osadzenie grafiki odbywa się już na finalnym dokumencie PDF.

7.1.11. Integracje

Oprócz realizacji własnej logiki dziedzinowej, aplikacja komunikuje się z usługami i odbiorcami spoza systemu. Zależności wychodzące – czyli wywołania usług zewnętrznych – są celowo umieszczone w warstwie infrastruktury i ukryte za abstrakcjami zdefiniowanymi w warstwie aplikacji, zgodnie z zasadą odwrócenia zależności przyjętą w całym rozwiązaniu (punkty *7.1.1* i *7.1.4*). Dzięki temu logika aplikacyjna posługuje się intencją operacji – na przykład „podłącz domenę” czy „odśwież katalog modeli” – bez wchodzenia w szczegóły protokołu oraz dostawcy. Sposób obsługi niepowodzeń zależy przy tym od charakteru integracji i opisano go przy poszczególnych przypadkach.

W kolejnych podrozdziałach opisano trzy integracje o różnym charakterze i kierunku. Integracja z dostawcą sieci brzegowej Cloudflare jest wychodząca i z poziomu aplikacji steruje infrastrukturą sieciową, podłączając wypożyczalniom ich własne domeny (podpunkt *7.1.11.1*). Integracja z publicznym katalogiem vPIC agencji NHTSA również jest wychodząca, lecz służy zasileniu danych referencyjnych – słownika marek i modeli pojazdów (podpunkt *7.1.11.2*). Trzeci przypadek ma kierunek odwrotny: nie jest zależnością od obcej usługi, lecz artefaktem udostępniającym własne API platformy jej konsumentom – typowanym klientom SDK generowanym z kontraktu (podpunkt *7.1.11.3*).

7.1.11.1. Cloudflare – podłączanie własnych domen wypożyczalni

Platforma Wellhicle udostępnia każdej wypożyczalni stronę publiczną pod subdomeną systemową, a opcjonalnie również pod jej własną domeną. Podłączenie własnej domeny wymaga dwóch działań po stronie dostawcy sieci brzegowej Cloudflare: zarejestrowania domeny jako *custom hostname* w mechanizmie *SSL for SaaS*, co obejmuje weryfikację własności metodą DCV (ang. *Domain Control Validation*) [120] oraz wystawienie certyfikatu TLS [121], a także dopisania reguły routingu (*ingress*) w tunelu Cloudflare, która kieruje ruch z nowej domeny do klastra. Niniejszy podrozdział dotyczy aplikacyjnej integracji, która rekonfiguruje ten tunel w czasie działania systemu i zarządza cyklem życia *custom hostname*. Wymagania funkcjonalne podłączania, zmiany i odłączania domeny zebrano w podrozdziale *5.3.1*(WF-ZWY-10/11), a model danych wypożyczalni – w podpunkcie *6.3.3.1*.

Komunikację z API Cloudflare realizuje dedykowana usługa infrastruktury, która wywołuje jego punkty końcowe HTTP i autoryzuje żądania tokenem dostępowym. Identyfikatory konta, tunelu i strefy oraz token są pobierane z konfiguracji środowiska i przekazywane jako sekrety podczas wdrożenia (podrozdział *7.4.1*). Nieudane odpowiedzi API są obsługiwane poprzez zwracanie obiektu Result.Failure z kodem błędu *CLOUDFLARE_API_ERROR*, co pozwala na traktowanie ich jako standardowych przypadków błędów oczekiwanych (wartość zwrotna), a nie wyjątków (podrozdział *7.1.6*). Usługa ta pełni dwie główne funkcje: zarządza konfiguracją tunelu Cloudflare poprzez pobieranie i aktualizowanie listy reguł ingress [122] oraz obsługuje niestandardowe domeny (tworzenie, pobieranie szczegółów i usuwanie) [123].

Logikę podłączania spina aplikacyjny serwis *DomainConfigurationService*, który integruje walidację, wywołania do Cloudflare oraz zapis stanu w agregacie wypożyczalni. Walidacja poprzedzająca operację sprawdza zgodność domeny z regułami domenowymi, jej globalną unikalność na platformie oraz obecność wymaganego rekordu CNAME wskazującego na subdomenę systemową wypożyczalni. Proces podłączenia przebiega sekwencyjnie: serwis pobiera aktualną konfigurację tunelu, dodaje regułę *ingress* łączącą nową domenę z wewnętrznym kontrolerem ruchu klastra, zapisuje zmiany w konfiguracji, tworzy niestandardową nazwę hosta (*custom hostname*), pobiera jego szczegóły i mapuje je na obiekt zawierający rekordy DNS do skonfigurowania przez właściciela (weryfikacja własności oraz SSL). Odłączenie domeny następuje odwrotnie – reguła *ingress* jest usuwana z konfiguracji, a *custom hostname* kasowany.

Istotną cechą tej orkiestracji jest ręczna kompensacja częściowo wykonanej operacji. Podłączenie składa się z kilku następujących po sobie wywołań zewnętrznego API, które nie tworzą operacji atomowej, dlatego po niepowodzeniu któregokolwiek z dalszych kroków serwis jawnie cofa skutki poprzednich: przywraca poprzednią konfigurację tunelu i usuwa utworzony wcześniej *custom hostname*. Dzięki temu nieudane podłączenie nie pozostawia po sobie osieroconej reguły *ingress* ani wpisu *hostname* po stronie Cloudflare. Fragment realizujący tę sekwencję wraz z kompensacją przedstawiono na *Listing 33*.

Listing 33 Fragment metody `AddDomainAsync` z klasy `DomainConfigurationService`, z projektu `Wellhicle.Application`. Źródło: opracowanie własne.

```
public async Task<Result<(DomainConfigDto, string)>>
AddDomainAsync(string domain, string subdomain, CancellationToken
cancellationToken)
{
    var targetHostname = domain.Trim().ToLowerInvariant();
    var httpHostHeader = subdomain.Trim().ToLowerInvariant() +
TenantRules.SubdomainSuffix;
    //...
    ingress.Add(new IngressRuleDto
    {
        Hostname = targetHostname,
        Service = ServiceName,
        OriginRequest = new CloudflareOriginRequestDto { HttpHostHeader =
httpHostHeader }
    });

    var newConfig = new TunnelConfigDto
    {
        Ingress = ingress
    };

    var putConfigurationResult = await
_cloudflareService.PutTunnelConfigurationAsync(newConfig,
cancellationToken);
    if (putConfigurationResult.IsFailure)
    {
```

```
        return putConfigurationResult.ToResult<(DomainConfigDto,
string)>();
    }

    var createCustomHostnameResult = await
_cloudflareService.CreateCustomHostnameAsync(targetHostname,
cancellationToken);
    if (createCustomHostnameResult.IsFailure)
    {
        await
_cloudflareService.PutTunnelConfigurationAsync(currentConfigurationResult
.Value.Config, cancellationToken);
        return createCustomHostnameResult.ToResult<(DomainConfigDto,
string), string>();
    }
    var customHostnameId = createCustomHostnameResult.Value!;

    var getCustomHostnameResult = await
_cloudflareService.GetCustomHostnameDetailsAsync(customHostnameId,
cancellationToken);
    if (getCustomHostnameResult.IsFailure)
    {
        await
_cloudflareService.PutTunnelConfigurationAsync(currentConfigurationResult
.Value.Config, cancellationToken);
        await
_cloudflareService.DeleteCustomHostnameAsync(customHostnameId,
cancellationToken);
        return getCustomHostnameResult.ToResult<(DomainConfigDto,
string), CustomHostnameDetailsResult>();
    }
    //...
}
```

Status weryfikacji domeny nie jest ustalany jednorazowo, lecz odświeżany cyklicznie w tle. Mechanizm Hangfire (opisany szczegółowo w podpunkcie *7.1.8.1*) uruchamia proces pobierający aktywne wypożyczalnie z przypisaną domeną, sprawdza ich aktualny status w Cloudflare i aktualizuje znacznik weryfikacji w agregacie. Administrator platformy może również wymusić tę synchronizację na żądanie. Domenę uznaje się za zweryfikowaną dopiero po osiągnięciu aktywnego statusu zarówno dla weryfikacji własności, jak i SSL.

Integracja jest powiązana z resztą backendu przez agregat wypożyczalni, który przechowuje identyfikator *custom hostname* oraz znacznik weryfikacji domeny (podpunkt *7.1.2.3*), a także przez rodzinę kodów błędów *CLOUDFLARE_** i *DOMAIN_** mapowanych na odpowiedzi API w sposób opisany w podrozdziale *7.1.6*. Przebieg podłączania domeny od strony użytkownika – wraz z prezentacją wymaganych rekordów DNS i kolejnymi stanami weryfikacji – modeluje diagram aktywności przypadku użycia UC-ZWY-05.

7.1.11.2. Katalog modeli pojazdów (vPIC NHTSA)

Nazwy marek i modeli pozyskiwane są z publicznego API vPIC (ang. Vehicle Product Information Catalog) agencji NHTSA [124]. Są to dane słownikowe uzupełniane na żądanie, poza ścieżką żądanie-odpowiedź użytkownika; zasilają katalog modeli wykorzystywany przy definiowaniu floty – tworzeniu grup pojazdów i ofert wypożyczalni.

Komunikację realizuje typowany HttpClient, wstrzykiwany jak pozostałe usługi infrastruktury (podrozdziały *7.1.1* i *7.1.4*), który odpytuje dwa punkty końcowe katalogu – listę marek oraz listę

modeli dla marki – z prostym ponawianiem prób przy błędach przejściowych. Serwis odświeżania mapuje nazwy marek przyjęte w systemie na ich odpowiedniki w vPIC (z normalizacją i dopasowaniem rozmytym, raportując marki nierozpoznane), pobiera dla nich modele i zapisuje pary marka-model w trybie upsert: dopisywane są wyłącznie pary nowe, więc ponowne uruchomienie nie powiela słownika. Wynikiem jest rekord statystyk pozbawiony danych wrażliwych; po wyczerpaniu prób całość kończy się ogólnym niepowodzeniem, bez typowanych błędów oczekiwanych z podrozdziału *7.1.6*, inaczej niż integracja z Cloudflare (podpunkt *7.1.11.1*).

Odświeżanie wyzwala wyłącznie administrator platformy, komendą z autoryzacją administracyjną (podpunkt *7.1.3.2*); katalog nie jest odświeżany cyklicznie, ponieważ dane słownikowe zmieniają się rzadko.

7.1.11.3. Klient SDK API (Wellhicle.Client / Kiota)

Aplikacja udostępnia własne REST API nie tylko jako punkt końcowy HTTP, lecz także jako gotowy, typowany pakiet kliencki (NuGet Wellhicle.Client). Dla odbiorcy API oznacza to, że zamiast samodzielnie kodować wywołania, serializację i nagłówki, sięga po bibliotekę odwzorowującą całe API w kodzie – co istotnie upraszcza pisanie integracji. Jest to przypadek o odwrotnym kierunku niż integracje opisane w podpunktach *7.1.11.1* i *7.1.11.2*: nie zależność od obcej usługi, lecz artefakt budowany z własnego kontraktu i przeznaczony dla konsumentów tego API. Cechą wyróżniającą jest to, że klient nie jest pisany ręcznie, tylko generowany z kontraktu.

Kontraktem jest pełny dokument OpenAPI budowany w warstwie prezentacji backendu i eksportowany bezpośrednio z działającej aplikacji. Przepływ GitHub Actions uruchamia skrypt budujący, który wykonuje te operacje sekwencyjnie: eksport kontraktu przez test integracyjny, generację klienta przy użyciu Kiota oraz spakowanie biblioteki do pakietu NuGet publikowanego w prywatnym rejestrze z oznaczeniem wersji. Z dokumentu OpenAPI narzędzie Kiota generuje komplet klas klienta – konstruktory żądań dla poszczególnych tras oraz modele danych. Ponieważ kontrakt pochodzi bezpośrednio z API i stanowi źródło prawdy, wygenerowany klient nie może rozbiec się z rzeczywistymi trasami ani modelami – każda zmiana w API ujawnia się w kliencie po ponownej generacji. Wygenerowana warstwa, licząca kilkaset plików, nie jest wersjonowana w repozytorium, lecz odtwarzana przy budowaniu pakietu.

Na wygenerowanym kliencie zespół utrzymuje jedynie cienką, ręcznie pisaną nakładkę: fabrykę tworzącą klienta, obiekt konfiguracji (adres API, poświadczenia, token dostępu) oraz komponent odpowiedzialny za uwierzytelnianie, który dodaje do każdego żądania nagłówki `Authorization: Bearer`, `X-Frontend-Source` i `X-Tenant-Subdomain`. Te same nagłówki rozpoznaje backend przy autoryzacji żądań z różnych aplikacji klienckich i kierowaniu ich do właściwej wypożyczalni (podrozdział *7.1.5*), więc klient posługuje się dokładnie tym samym kontraktem co pozostali konsumenci API. Nakładka ujednolica też obsługę błędów, dzięki czemu po stronie konsumenta błędy są rozpoznawane tak samo jak w backendzie.

Najważniejszym realnym odbiorcą pakietu jest integracja z systemem ERP enova365 (podrozdział *7.5*), która – w przeciwieństwie do ręcznego kodowania wywołań – buduje swój łącznik na podstawie gotowego klienta, co znacząco ułatwiło jej implementację. Należy jednak zaznaczyć, że ten gotowy klient nie został wykorzystany przez aplikacje przeglądarkowe ani przez aplikację mobilną – powstał stosunkowo późno, gdy te rozwiązania były już praktycznie ukończone.

Należy przy tym odróżnić dwie rzeczy. Opisywany pakiet stanowi klienta wewnętrznego API platformy – tego samego, z którego korzystają jej własne aplikacje – i to on ułatwił budowę integracji takich jak enova365. Czym innym byłoby udostępnienie systemu zewnętrznym integratorom: wymagałoby ono zaprojektowania i napisania osobnego, dedykowanego rozwiązania wystawiającego publiczny zestaw punktów końcowych API – być może pokrywający się z obecnym, a być może celowo

węższy i inaczej ukształtowany (stabilny, wersjonowany kontrakt o zawężonych uprawnieniach) – wraz z odpowiadającym mu klientem. Opisywany SDK takiej roli nie pełni i nie był pod nią projektowany; jest to kierunek ewentualnych dalszych prac, a nie brak w bieżącym zakresie.

7.2. Interfejs przeglądarkowy

Implementacja poszczególnych aplikacji przeglądarkowych składających się na system Wellhicle opiera się na zestawie narzędzi wykorzystanych do ich zbudowania, mechanizmach zapewniających bezpieczeństwo oraz bibliotece elementów współdzielonych pomiędzy aplikacjami.

7.2.1. Bezpieczeństwo

Mechanizmy odpowiadające za bezpieczeństwo aplikacji przeglądarkowych zaimplementowano w więcej niż jednej aplikacji, lecz nie wydzielono ich jako zasobów współdzielonych, ponieważ ich zakres i konfiguracja różnią się w zależności od funkcji realizowanych przez daną aplikację. Każdy z mechanizmów przedstawiono na konkretnym przykładzie, wskazując na końcu, czym różni się w zależności od aplikacji.

7.2.1.1. Uwierzytelnianie

Uwierzytelnianie w aplikacji do zarządzania wypożyczalnią przebiega w sposób standardowy. Serwer zwraca w treści odpowiedzi obiekt zawierający m.in. token dostępu (ang. *access token*) oraz token odświeżania (ang. *refresh token*). Po uwierzytelnieniu token dostępu jest wymagany przez serwer w nagłówku żądania. Wybór miejsca przechowywania tokenu po stronie klienta wymagał uwzględnienia ograniczeń bezpieczeństwa. Przechowywanie tokenu w pamięci lokalnej (ang. *local storage*) lub pamięci sesji (ang. *session storage*) naraża aplikację na kradzież tokenu w wyniku ataku XSS (*Cross-Site Scripting*), ponieważ skrypty działające w kontekście strony mają do nich pełny dostęp [125]. Z tego względu zdecydowano się na przechowywanie danych autoryzacyjnych wyłącznie w pamięci operacyjnej jako prywatne elementy klasy serwisu. Rozwiązanie to eliminuje ryzyko odczytu tokenu przez złośliwe skrypty, ponieważ dane przechowywane w ten sposób nie są dostępne poza instancją serwisu [126]. Wadą tego podejścia jest utrata sesji po odświeżeniu strony, co uznano za akceptowalny kompromis w kontekście wymagań aplikacji.

Identyfikator wypożyczalni odczytywany jest z tokenu dostępu przez zdekodowanie po stronie klienta. Operacja ta jest bezpieczna, ponieważ integralność danych jest gwarantowana przez serwer. Zmodyfikowany token zostanie przez niego odrzucony podczas weryfikacji.

Identyczne rozwiązanie zastosowano we wszystkich aplikacjach, z wyjątkiem aplikacji do prezentacji oferty systemu, która nie wymaga uwierzytelnienia.

7.2.1.2. Autoryzacja

Autoryzację w aplikacji do zarządzania wypożyczalnią zaimplementowano jako *interceptor* HTTP. Mechanizm ten pozwala na centralne modyfikowanie żądań, np. dołączanie nagłówków autoryzujących bez konieczności powielania tej logiki w serwisach. Na *Listing 34* zostało przedstawione działanie *interceptora*. Obsługuje ono trzy przypadki:

- Pierwszy dotyczy żądań oznaczonych tokenem kontekstu HTTP. Token ten jest ustawiany ręcznie. Żądania oznaczone w ten sposób nie wymagają autoryzacji.

- Drugi przypadek obejmuje żądania, dla których token dostępu wygasł. Przyjęto podejście, w którym token uznawany jest za wygasły dokładnie w momencie, gdy jego czas ważności minął, bez wyprzedzającego marginesu czasowego. W takiej sytuacji interceptor podejmuje próbę odświeżenia sesji, zanim żądanie zostało wysłane.
- Trzeci przypadek dotyczy żądań wysyłanych z aktywnym tokenem dostępu. Jeśli serwer mimo to zwróci odpowiedź z kodem błędu HTTP 401, interceptor podejmuje próbę odświeżenia sesji i ponowienia żądania, co przedstawiono na *Listing 35*. Wyjątkiem jest sytuacja, gdy błąd występuje przy próbie odświeżenia sesji, wówczas dane sesji są usuwane, a użytkownik przekierowywany jest na ekran logowania.

Listing 34 Funkcja przechwytująca żądania. Źródło: opracowanie własne.

```
export const authInterceptor: HttpInterceptorFn = (request:
HttpRequest<unknown>, next: HttpHandlerFn):
Observable<HttpEvent<unknown>> => {
    const authService = inject(AuthService);
    if (request.context.get(authSkipped)) {
       return next(request);
    }
    if (authService.hasAccessTokenExpired()) {
       return refreshAndRetry(request, next);
    }
    request = setHeaders(request, authService.getAccessToken()!);
    return next(request).pipe(
       catchError((error) => {
          if (!(error instanceof HttpErrorResponse && error.status ===
401)) {
             return throwError(() => error);
          }
          return refreshAndRetry(request, next);
       }),
    );
};
```

Listing 35 Metoda obsługująca żądanie odświeżenia sesji. Źródło: opracowanie własne.

```
refresh(refreshToken: string): Observable<AuthData> {
    return this.http.post<AuthData>(`${this.url}/refresh`, { refreshToken
}, {
       context: new HttpContext().set(authSkipped, true),
    }).pipe(
       tap((authData) => this.setAuthStorage(authData)),
       catchError((error) => {
          this.clearAuthStorage();
          return throwError(() => error);
       }),
    );
}
```

Mechanizm autoryzacji został zastosowany we wszystkich aplikacjach z wyjątkiem aplikacji do prezentacji oferty systemu. We wszystkich przypadkach przyjęto jednolitą implementację. W przypadku przechwytywania różnicę stanowi jedynie zbiór żądań, a w przypadku kontroli dostępu objęte nią widoki.

7.2.1.3. Kontrola dostępu do widoku

Kontrola dostępu do widoków jest uzupełnieniem mechanizmu autoryzacji. W frameworku Angular jest uzyskiwana na poziomie *routera* przy pomocy mechanizmu *route guard* typu

`CanActivate`. Mechanizm ten weryfikuje obecność tokenu dostępu, a w przypadku jego braku przekierowuje użytkownika do widoku logowania.

We wszystkich aplikacjach, poza aplikacją do prezentacji oferty systemu, która nie wymagała autoryzacji, mechanizm działa analogicznie. Różnicę stanowią weryfikowany warunek oraz reakcja na jego niespełnienie.

7.2.2. Biblioteka elementów powszechnie używanych w aplikacjach

Biblioteka @wellhicle/common gromadzi elementy powszechnie używane, współdzielone pomiędzy aplikacjami przeglądarkowymi. Wydzielenie ich do wspólnego pakietu pozwoliło wyeliminować powielanie kodu oraz zapewnić spójność w całym systemie. Bibliotekę zrealizowano przy pomocy Angulara, korzystając z narzędzi takich jak Angular CLI i ng-packagr. Opublikowano ją w usłudze GitHub Packages jako pakiet prywatny, co ograniczyło dostęp wyłącznie do uprawnionych członków organizacji Wellhicle na GitHubie. Aplikacje traktują bibliotekę analogicznie do innych zależności, instalując ją przez menedżera pakietów npm.

Biblioteka zawiera również elementy, którym ze względu na ich prostotę nie poświęcono osobnych podpunktów: obiekty dostarczające tłumaczenia etykiet komponentów Angular Material, pliki tłumaczeń, metody pomocnicze, wspólne obiekty modelu oraz walidatory.

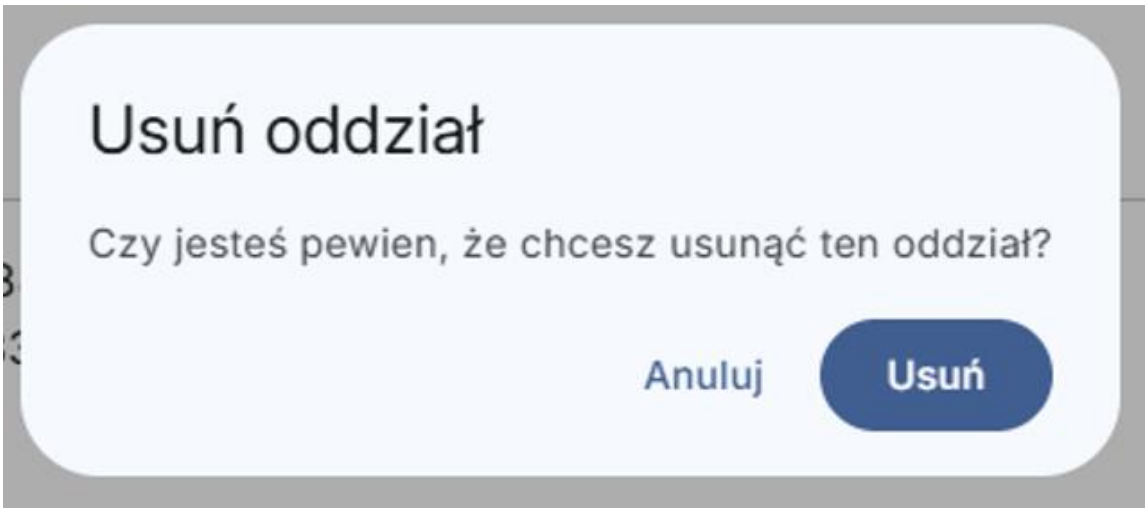
7.2.2.1. Okno potwierdzenia akcji

Niektóre akcje w aplikacji są nieodwracalne lub niosą daleko idące skutki, dlatego wymagają dodatkowego potwierdzenia przez użytkownika. Aby to zrealizować stworzono generyczny komponent potwierdzenia akcji oparty na *dialog* z Angular Material, który przyjmuje konfigurację przedstawioną na *Listing 36*.

Listing 36 Konfiguracja okna potwierdzenia akcji. Źródło: opracowanie własne.

```
export interface ConfirmDialogConfig {
    title: string;
    content: string;
    action: string;
}
```

Przykładowe okno potwierdzenia akcji usunięcia oddziału zostało przedstawione na *Rysunek 97*.



Rysunek 97 Okno potwierdzenia akcji usunięcia oddziału. Źródło: opracowanie własne.

Etykieta przycisku anulowania jest tłumaczona wewnątrz komponentu, ponieważ pozostaje niezmienna niezależnie od akcji.

7.2.2.2. Komponent obsługujący kwoty

Komponent obsługujący kwoty odpowiada za przyjmowanie oraz prezentowanie kwot w formacie zależnym od waluty oraz ustawień regionalnych. Wprowadzenie go wynikało z dwóch wymagań. Po pierwsze, kwoty przechowywane są w najmniejszej jednostce waluty, natomiast użytkownikowi powinny być prezentowane w jednostce głównej. Po drugie, system obsługuje wiele walut oraz aplikacja jest prezentowana w wielu językach, dlatego sposób formatowania kwoty zależy zarówno od ustawionej waluty, jak i od preferencji dotyczących języka. Komponent zrealizowano jako własną kontrolkę pola formularza opartą na komponencie input z Angular Material, zgodnie ze wzorcem opisanym w dokumentacji [127]. Przykładowe pole wprowadzania kwoty przedstawiono na *Rysunek 98*.



Rysunek 98 Pole wprowadzania kwoty. Źródło: opracowanie własne.

Głównym założeniem komponentu jest rozdzielenie wartości przechowywanej od prezentowanej. Wartość przechowywana jest jako liczba całkowita w najmniejszej jednostce waluty, a użytkownikowi prezentowany jest sformatowany tekst. Przeliczanie i formatowanie w obu kierunkach opiera się na obiekcie `Intl.NumberFormat` skonfigurowanym dla danej waluty oraz aktywnych ustawień regionalnych. Przy prezentacji kwoty użytkownikowi wartość przeliczana jest na jednostkę główną i formatowana zgodnie z regułami wyżej wspomnianego obiektu. W kierunku odwrotnym wpisany tekst jest normalizowany z wykorzystaniem separatora, a następnie przeliczany z powrotem na liczbę całkowitą, co przedstawiono na *Listing 37*.

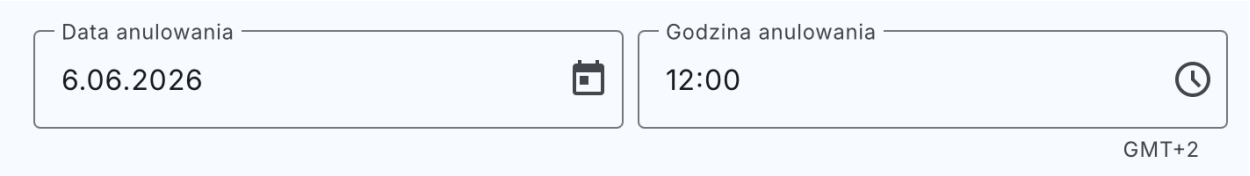
Listing 37 Metoda wyliczająca kwotę dla najmniejszej możliwej jednostki waluty. Źródło: opracowanie własne.

```
private minorUnitAmount(value: string, locale: string, currency: string):
number | null {
    const numberFormat = new Intl.NumberFormat(locale, {
       style: "currency",
       currency: currency,
    });
    const parts = numberFormat.formatToParts(1.1);
    const decimalPart = parts.find(part => part.type === "decimal");
    const decimalSeparator = decimalPart !== undefined ?
decimalPart.value : ".";
    let normalizedValue = value
       .replace(new RegExp(`[^0-9-${decimalSeparator}]`, "g"), "")
       .replace(decimalSeparator, ".");
    const floatValue = parseFloat(normalizedValue);
    if (isNaN(floatValue)) {
       return null;
    }
    const fractionDigits =
numberFormat.resolvedOptions().maximumFractionDigits ?? 0;
    const multiplier = Math.pow(10, fractionDigits);
    return Math.round(floatValue * multiplier);
}
```

Dopełnieniem komponentu jest *pipe* prezentujący kwoty w oparciu o te same reguły formatowania.

7.2.2.3. Adaptery dla komponentów wyboru daty i czasu

Dyrektywa `ZonedDateTimeDirective` konfigurująca komponenty *datepicker* oraz *timepicker* z Angular Material wykorzystuje typy udostępnione przez Temporal API. Wprowadzenie własnego adaptera dat wynikało z ograniczeń adaptera domyślnego, który operuje na wbudowanym typie `Date`. Typ ten nie obsługuje stref czasowych i nie zapewnia niezmienności, co w aplikacji operującej na zdarzeniach przypisanych do konkretnych stref czasowych generowało szereg trudnych do przewidzenia błędów [128]. Problemy te rozwiązuje Temporal API wprowadzając dedykowane typy, takie jak `Temporal.ZonedDateTime`. TypeScript 6 obsługuje Temporal API, jednak Angular 20 go nie wspiera [129]. Z tego względu zastosowano bibliotekę temporal-polyfill. Aby komponenty *datepicker* oraz *timepicker* mogły operować na typach z Temporal API, dyrektywa dostarcza im, poprzez wstrzykiwanie zależności, trzy elementy: adapter dat oparty na tych typach, formaty wyświetlania oraz tłumaczenia etykiet. Adapter odpowiada za tworzenie, porównywanie oraz formatowanie dat i czasu, a jego implementację oparto na wzorcu opisanym w dokumentacji Angular Material [130]. Dyrektywę można zastosować na polu złożonym, w którym komponenty *datepicker* oraz *timepicker* współdzielą jedną kontrolkę formularza, zgodnie z ich integracją opisaną w dokumentacji Angular Material [131]. Cechą charakterystyczną dyrektywy jest przyjmowanie identyfikatora strefy czasowej. Przykładowe pola wyboru daty i czasu w strefie czasowej oddziału wypożyczalni przedstawiono na *Rysunek 99*.



Rysunek 99 Pole wyboru daty i czasu w strefie czasowej oddziału wypożyczalni. Źródło: opracowanie własne.

Adapter rozdziela wartość wpisywaną przez użytkownika od wartości zwracanej. Tekst wprowadzany w polu przekształcany jest na obiekt `Temporal.ZonedDateTime`, co wymaga jego sparsowania. Rozdzielenie wpisanego tekstu po separatorze daje trzy liczby, lecz sposób przypisania ich do określonego roku, miesiąca i dnia zależy od ustawień regionalnych. Z tego względu kolejność składowych odczytywana jest z obiektu `Intl.DateTimeFormat`. Z tak rozpoznanych składowych tworzony jest obiekt, który odrzuca wartości spoza zakresu dopuszczalnego dla danej daty. Mechanizm ten przedstawiono na *Listing 38*.

Listing 38 Przekształcanie tekstu wprowadzonego przez użytkownika na obiekt daty. Źródło: opracowanie własne.

```
override parse(value: any, _: string): Temporal.ZonedDateTime | null {
    if (typeof value === "string") {
       const array = value.trim().match(/[0-9]+/g);
       if (array?.length !== 3) {
          return null;
       } else {
          const parts = new
Intl.DateTimeFormat(this.locale).formatToParts();
          const types = parts
             .filter(part => part.type === "year" || part.type ===
"month" || part.type === "day")
             .map(part => part.type);
          let year = 0, month = 0, day = 0;
          types.forEach((type, index) => {
             const number = Number(array[index]);
```

```
            if (type === "year") year = number;
            if (type === "month") month = number;
            if (type === "day") day = number;
         });
         try {
            return Temporal.ZonedDateTime.from({ year, month, day,
timeZone: this.timeZoneId }, { overflow: "reject" });
         } catch (error) {
            return null;
         }
      }
   } else if (value instanceof Temporal.ZonedDateTime) {
      return Temporal.ZonedDateTime.from(value);
   } else {
      return null;
   }
}
```

Analogiczny mechanizm zastosowano przy odczycie czasu. Format godziny również zależy od ustawień regionalnych, przy czym dodatkową zmienną jest stosowanie zapisu 12-godzinnego lub 24-godzinnego, rozpoznawanego na podstawie obiektu *`Intl.DateTimeFormat`*. W zapisie 12-godzinnym godzina przeliczana jest na podstawie oznaczenia przedpołudniowego (a.m.) lub popołudniowego (p.m.).

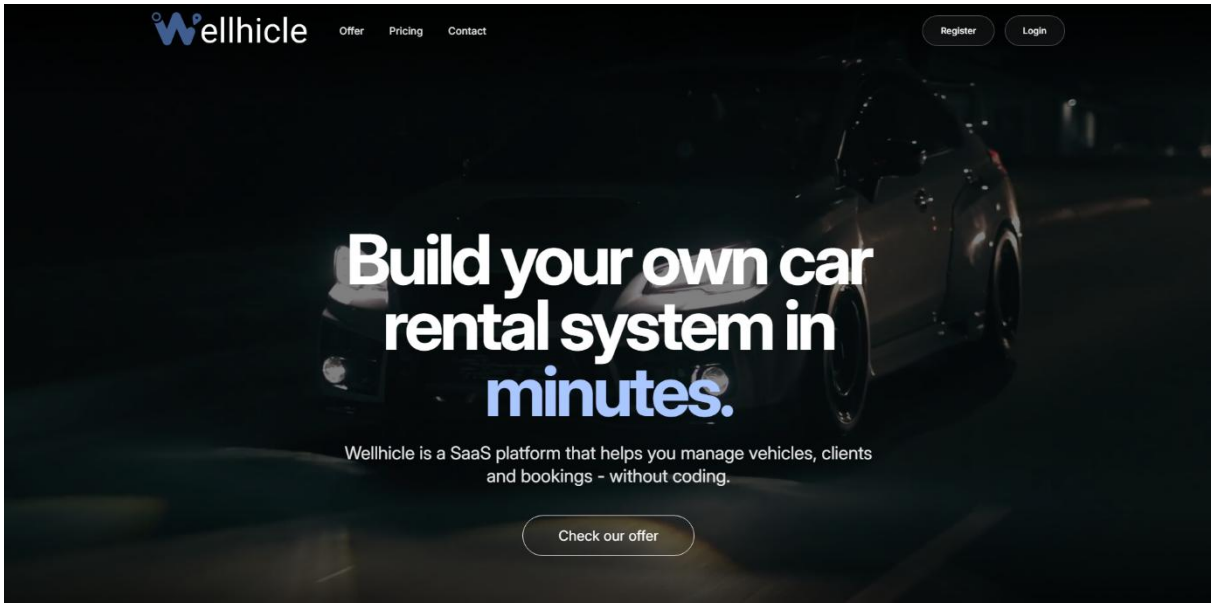
Analogiczne rozwiązanie zastosowano dla pozostałych typów takich jak *`Temporal.PlainDate`*. Całość uzupełniono *pipe'ami* formatującymi te typy do postaci prezentowanej w widokach.

7.2.3. Aplikacja do prezentacji oferty systemu

Aplikacja do prezentacji oferty systemu została zaimplementowana w oparciu o makiety przedstawione w *6.4.3.1*. Końcowa implementacja zachowuje ich podstawowy układ, jednak wprowadzono zmiany wizualne, które zostaną opisane w kolejnych podpunktach.

7.2.3.1. Sekcja główna strony

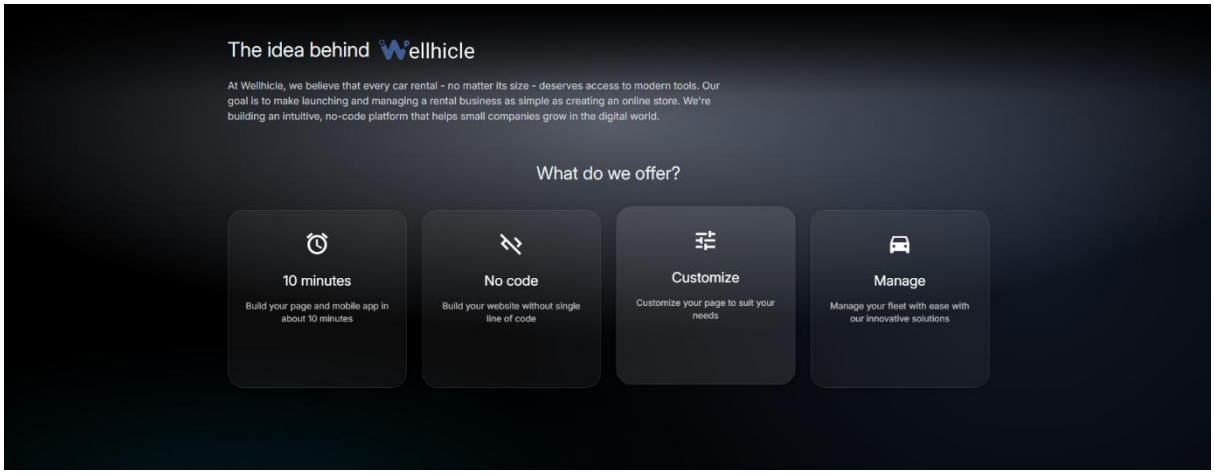
Do implementacji głównej sekcji strony wykorzystano wybrane komponenty Angular Material, w szczególności pasek narzędzi (ang. *toolbar*). Przyciski rejestracji i logowania przekierowują użytkownika do osobnej aplikacji. W stosunku do makiety zmieniono sposób prezentacji tła. Statyczne tło zastąpiono zapętlonym filmem. Sekcja została zaprezentowana na *Rysunek 100.*



Rysunek 100 Widok sekcji głównej strony publicznej. Źródło: opracowanie własne.

7.2.3.2. Sekcja informacyjna strony

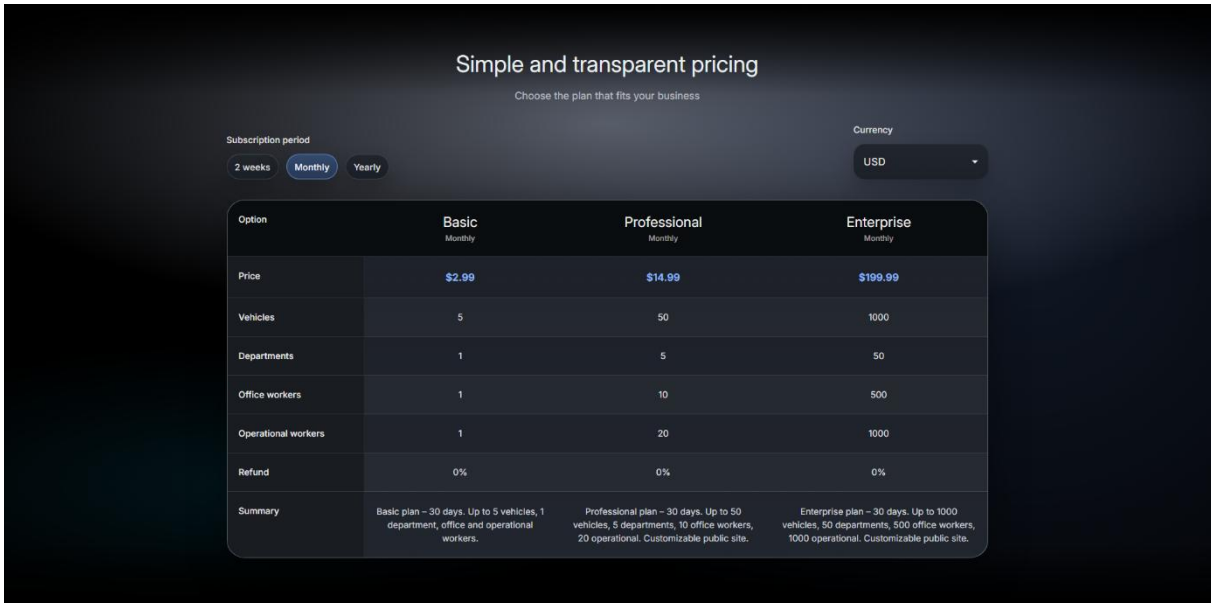
Do implementacji sekcji informacyjnej przedstawionej na *Rysunek 101,* wykorzystano komponent *card* z Angular Material. W stosunku do makiety zachowano ogólny układ sekcji, jednak dostosowano wygląd kart do ciemnej stylistyki aplikacji.



Rysunek 101 Widok sekcja informacyjnej strony publicznej. Źródło: opracowanie własne.

7.2.3.3. Sekcja planów

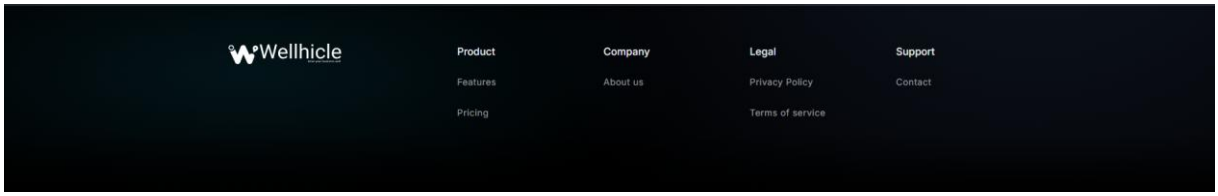
Największe różnice między makietą a implementacją występują w sekcji planów przedstawionej na *Rysunek 102* Zmiana wynikała z potrzeby prezentowania planów pobieranych z API. Zastosowanie tabeli zamiast kart pozwoliło lepiej przedstawić większą liczbę parametrów i ułatwiło bezpośrednie porównanie ofert.



Rysunek 102 Widok sekcji planów strony publicznej. Źródło: opracowanie własne.

7.2.3.4. Stopka

Bezpośrednio po sekcji prezentacji oferty, która stanowi ostatnią na stronie, umieszczono stopkę przedstawioną na *Rysunek 103.*



Rysunek 103 Stopka strony publicznej. Źródło: opracowanie własne.

7.2.4. Aplikacja do zarządzania wypożyczalnią

Implementację warstwy przeglądarkowej aplikacji do zarządzania wypożyczalnią omówiono ze szczególnym uwzględnieniem rozwiązań przyjętych na etapie implementacji: zastosowanych komponentach, sposobie realizacji kluczowych mechanizmów oraz różnicach względem wcześniej przygotowanych makiet wraz z ich uzasadnieniem. Opis podzielono na mechanizmy wspólne dla wielu widoków oraz implementację poszczególnych widoków, przy czym część z nich omówiono na wybranych przykładach.

7.2.4.1. Rejestracja wypożyczalni

Proces rejestracji wypożyczalni obejmuje trzy następujące po sobie widoki: rejestracja, logowanie oraz kreator wypożyczalni. Wyżej wymienione widoki zostały zrealizowane zgodnie z założeniami, przy czym końcowa implementacja różni się od wcześniej przygotowanych makiet. Różnice te zostały opisane w dalszej części tego podpunktu.

Rejestracja i logowanie

Widoki rejestracji oraz logowania zostały oparte na makietach z podpunktu *6.4.3.2*. Oba widoki zostały zrealizowane z wykorzystaniem komponentu c*ard* z biblioteki Angular Material. Na etapie implementacji dodano możliwość podglądu wpisanego hasła w polach formularza, co przedstawiono na *Rysunek 104* oraz *Rysunek 105*.



Rysunek 104 Widok rejestracji właściciela wypożyczalni. Źródło: opracowanie własne.



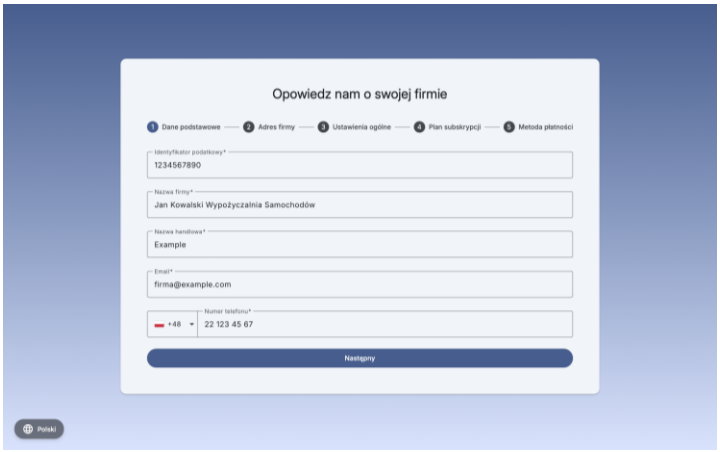
Rysunek 105 Widok logowania pracownika wypożyczalni. Źródło: opracowanie własne.

W ramach procesu logowania zastosowano mechanizm uwierzytelnienia opisany w podpunkcie *7.2.1.1,* którego wynikiem jest przekierowanie do kreatora wypożyczalni. W celu ochrony tego widoku przed dostępem osoby nieautoryzowanej zastosowano mechanizm *route guard* typu *CanActivate*, który został opisany dokładnie w podpunkcie *7.2.1.2*.

Kreator wypożyczalni

Widok kreatora wypożyczalni został oparty na makiecie przedstawionej w podpunkcie *6.4.3.3*. Końcowa implementacja jest jednak znacznie bardziej rozbudowana niż pierwotny projekt. Kreator obejmuje pięć kroków zamiast trzech przewidzianych w makiecie. Rozszerzenie to wynikało z doprecyzowania wymagań funkcjonalnych w trakcie realizacji projektu. Do implementacji kreatora wykorzystano komponent *stepper* z biblioteki Angular Material, który nie został uwzględniony w makiecie.

Pierwszy krok kreatora obejmuje formularz danych podstawowych firmy. Do obsługi pól tekstowych zastosowano komponenty *form field* z Angular Material. Natomiast dla bardziej zaawansowanego pola numeru telefonu zdecydowano się na gotowy komponent z innej biblioteki, co przedstawiono na *Rysunek 106*.



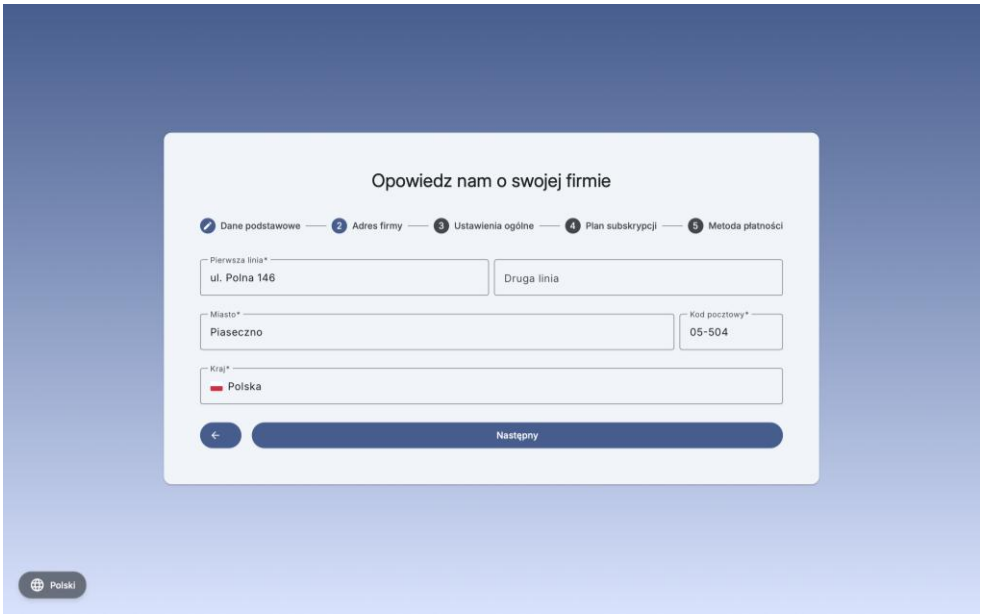
Rysunek 106 Widok kreatora wypożyczalni – krok 1. Źródło: opracowanie własne.

Ngx-material-intl-tel-input to biblioteka udostępniająca gotowy komponent do wprowadzania międzynarodowych numerów telefonów. Rozwiązanie integruje listę wyboru kraju oraz zapewnia wbudowane formatowanie i walidację numerów telefonów [132]. Wybór tej konkretnej biblioteki uzasadniają czynniki:

- Komponent zbudowano na bazie Angular Material, co zapewnia spójność z pozostałymi wbudowanymi komponentami.
- Biblioteka korzysta z systemu tokenów Angular Material, dzięki czemu możliwe jest dostosowanie jej wyglądu do przyjętego motywu aplikacji.
- Do walidacji numerów telefonów wykorzystano bibliotekę google-libphonenumber, która została zastosowana również po stronie serwera.
- Biblioteka obsługuje dynamiczną zmianę nazw krajów, co jest zgodne z założeniami dotyczącymi wielojęzyczności systemu.

Samodzielna implementacja analogicznego komponentu wymagałaby znacznego nakładu pracy, dlatego zdecydowano się na gotowe rozwiązanie.

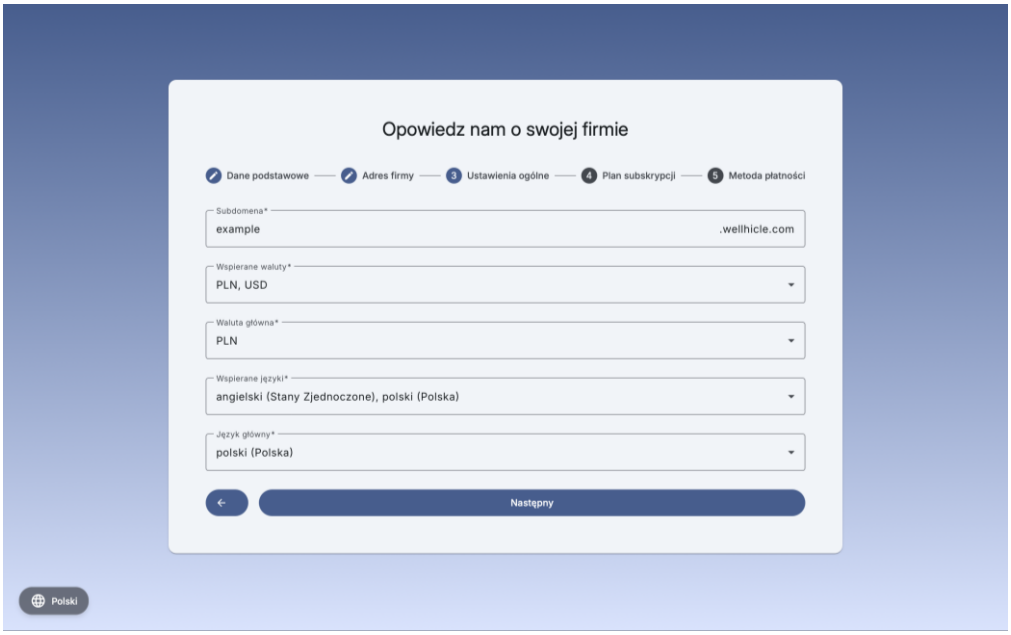
Drugi krok kreatora obejmuje formularz z adresem firmy. Krok ten wyodrębniono z pierwotnego pierwszego kroku makiety, który łączył dane podstawowe firmy z danymi adresowymi, co przedstawiono na *Rysunek 107*. Podział uzasadniono nadmierną liczbą pól formularza prezentowanych jednocześnie na jednym ekranie.



Rysunek 107 Widok kreatora wypożyczalni – krok 2. Źródło: opracowanie własne.

Jedynym niestandardowym elementem tego kroku jest pole wyboru kraju. Zdecydowano się na implementację własną, opartą na komponencie *autocomplete* z biblioteki Angular Material. Komponent przyjmuje model kraju przychodzący z serwera i obsługuje dynamiczną zmianę języka nazw krajów zgodnie z aktywnym ustawieniem języka aplikacji. Rezygnacja z gotowej biblioteki wynikała z braku dostępnego rozwiązania spełniającego podstawowe warunki: obsługę dynamicznego tłumaczenia oraz wystarczające wsparcie społeczności gwarantujące długoterminowe utrzymanie pakietu. Implementacja własna wymagała jedynie integracji z biblioteką flag-icons dostarczającej ikony flag państw, co ograniczyło nakład pracy do minimum.

Trzeci krok kreatora obejmuje konfigurację ustawień ogólnych wypożyczalni, dostępnych później do podglądu i edycji w widoku ustawień opisanym w podpunkcie *7.2.4.12*. W stosunku do makiety zmieniono sposób wyboru obsługiwanych języków. Zamiast komponentu *chips* zastosowano *select*, co przedstawiono na *Rysunek 108*. Decyzja została podyktowana potencjalnym rozszerzeniem liczby dostępnych języków w przyszłości, dla której *chips* mogłyby zajmować zbyt dużo miejsca.



Rysunek 108 Widok kreatora wypożyczalni – krok 3. Źródło: opracowanie własne.

Wybrane w tym kroku języki, a także waluty, wyznaczają zestaw zakładek prezentowanych w formularzach wielojęzycznych i wielowalutowych aplikacji, co zaprezentowano na przykładzie widoku w podpunkcie *7.2.4.7*.

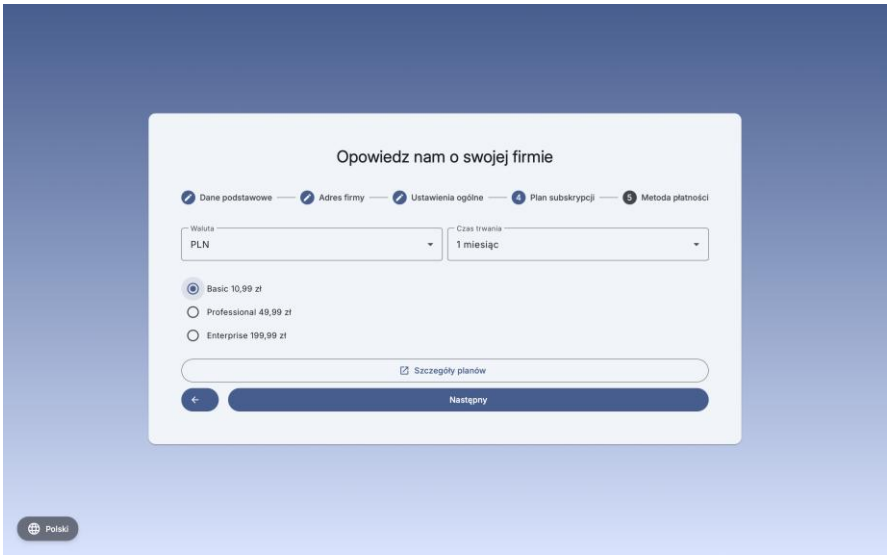
Elementem wymagającym niestandardowego podejścia było pole subdomeny. Reguły walidacji po stronie klienta odpowiadają regułom stosowanym przez serwer, przy czym minimalna długość subdomeny jest weryfikowana synchronicznie po stronie klienta, aby nie obciążać serwera żądaniami w przypadkach, które można odrzucić lokalnie [133]. Weryfikacja pozostałych reguł, takich jak sprawdzenie, czy dana subdomena jest już w użyciu lub czy nie widnieje na liście słów niedozwolonych, wymaga odpytania serwera. Z tego względu zastosowano walidator asynchroniczny (ang. *asynchronous validator*), co przedstawiono na *Listing 39*. Walidator skonfigurowano dodatkowo tak aby żądanie było wysyłane dopiero po opuszczeniu pola (ang. *blur*) przez użytkownika, a nie po każdym naciśnięciu klawisza.

W toku współpracy zespołów *frontendowego* i *backendowego* przyjęto konwencję, zgodnie z którą błąd walidacji subdomeny jest zwracany z kodem HTTP 200, a wynik weryfikacji przekazywany jako obiekt w ciele odpowiedzi. Takie podejście oddziela błędy walidacji od błędów protokołu HTTP, co upraszcza obsługę po stronie klienta. Konwencja nazewnicza błędu walidatora została celowo ujednolicona ze stylem stosowanym przez wbudowane walidatory Angulara. Dzięki temu obsługa błędu subdomeny w szablonie przebiega w sposób spójny z pozostałymi polami formularza.

Listing 39 Funkcja asynchronicznie walidująca subdomenę. Źródło: opracowanie własne.

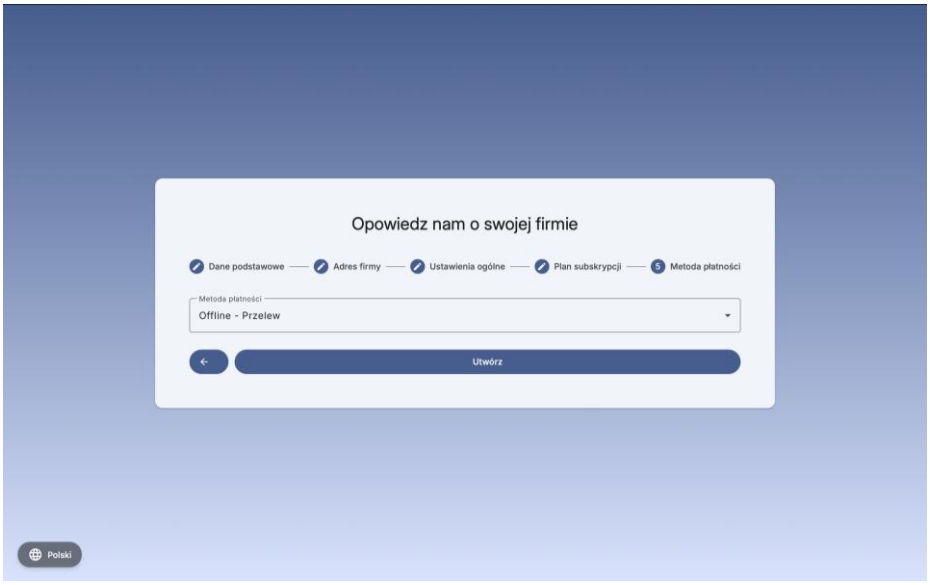
```
static subdomain(tenantService: TenantService): AsyncValidatorFn {
    return (control: AbstractControl): Observable<ValidationErrors |
null> => {
       return tenantService.validateSubdomain(control.value).pipe(
          map((validation) => (validation.isValid ? null : { subdomain:
validation })),
          catchError((error) => {
             if (error instanceof HttpErrorResponse) {
                if (isCustomError(error.error)) {
                   return of({ subdomain: { isValid: false, error:
error.error } });
                }
             }
             return of(null);
          }),
       );
    };
}
```

Czwarty krok kreatora obejmuje wybór planu. Widok udostępnia odnośnik do szczegółów planów znajdujących się na stronie aplikacji do prezentacji oferty systemu, przedstawionej w punkcie *7.2.3*. W stosunku do makiety wprowadzono dwa dodatkowe pola: wybór waluty oraz wybór okresu trwania planu, co przedstawiono na *Rysunek 109*. Wynikało to ze zmian wymagań funkcjonalnych, które nastąpiły po powstaniu makiet. Stwierdzono, że pakiety subskrypcji powinny być dostępne w wielu walutach oraz na różne okresy rozliczeniowe.



Rysunek 109 Widok kreatora wypożyczalni – krok 4. Źródło: opracowanie własne.

Piąty krok kreatora nie był uwzględniony w makietach. Został dodany w widoku kreatora wypożyczalni na późniejszym etapie realizacji projektu, po zaimplementowaniu modułu płatności przez serwer. Krok ten obejmuje wybór metody płatności za wybrany plan, co przedstawiono na *Rysunek 110*.



Rysunek 110 Widok kreatora wypożyczalni – krok 5. Źródło: opracowanie własne.

Po ukończeniu procesu rejestracji wypożyczalni następuje przeniesienie użytkownika do panelu głównego aplikacji. Dostęp do panelu głównego jest chroniony przez *route guard* typu *CanActivate*, który weryfikuje posiadanie identyfikatora tenanta, a w przypadku jego braku przekierowuje użytkownika do widoku kreatora wypożyczalni.

7.2.4.2. Kontrola dostępu oparta na rolach

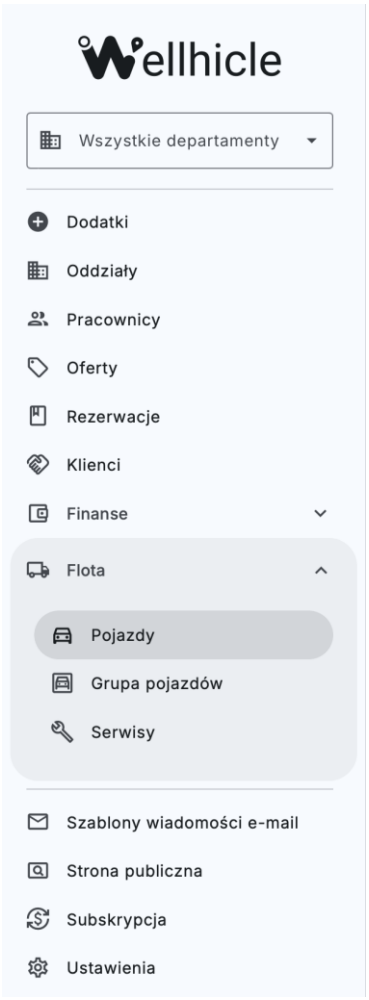
Mechanizm zarządzania uprawnieniami oparty na rolach – RBAC (ang. *Role-Based Access Control*) zastosowano w aplikacji do zarządzania wypożyczalnią, ponieważ korzystają z niej dwa typy użytkowników: właściciel oraz pracownik biurowy. Zabezpieczenie dostępu zrealizowano na dwóch poziomach: konfiguracji nawigacji i interfejsu użytkownika. Na poziomie konfiguracji nawigacji zastosowano mechanizm *route guard* typu *CanActivate*, który weryfikuje posiadanie roli właściciela i

w przypadku jej braku blokuje dostęp do widoku. Natomiast na poziomie interfejsu użytkownika elementy nawigacji są wyświetlane warunkowo w zależności od posiadania roli właściciela.

7.2.4.3. Nawigacja boczna i pasek narzędzi

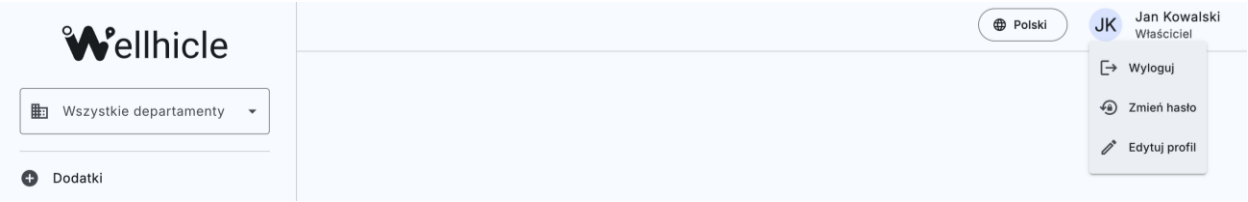
Nawigowanie po aplikacji zostało zrealizowane zgodnie z makietą przedstawioną w punkcie *6.4.3.4* za pomocą komponentu *sidenav* z Angular Material. Sekcje grupujące powiązane widoki, takie jak pojazdy, zbudowano przy użyciu komponentu *expansion panel* z Angular Material. Zostały one dostosowane stylistycznie do sekcji pojedynczych, co przedstawiono na *Rysunek 111*. Poszczególne pozycje nawigacji powiązano z trasami aplikacji za pomocą dyrektywy *RouterLink*. Sposób ukrywania pozycji w zależności od roli został przedstawiony w podpunkcie *7.2.4.2*.

Elementem nieuwzględnionym w makiecie jest lista rozwijana pozwalająca na wybór jednego lub wielu departamentów, umieszczona w nagłówku panelu nawigacyjnego. Ze względu na powielanie tego elementu jako filtra danych w kolejnych widokach list, wyodrębniono go jako element współdzielony. Komponent inicjalizuje dostępną listę oddziałów w zależności od roli zalogowanego użytkownika. Właściciel otrzymuje pełną listę oddziałów pobraną z serwera, natomiast pracownik biurowy widzi wyłącznie oddział przypisany do jego konta. Sposób działania tego komponentu został przedstawiony w podpunkcie *7.2.4.4*.



Rysunek 111 Widok nawigacji. Źródło: opracowanie własne.

Nawigację uzupełnia pasek narzędzi zrealizowany za pomocą komponentu toolbar z Angular Material, przedstawiony na *Rysunek 112*. Element ten nie został uwzględniony w makiecie. Przeniesiono do niego przełącznik języka wyświetlania oraz opcje profilu użytkownika. Decyzję tę podyktowała ograniczona przestrzeń panelu nawigacyjnego, w którym przy dużej liczbie pozycji elementy te mogłyby być niewidoczne bez przewijania.



Rysunek 112 Widok paska narzędzi. Źródło: opracowanie własne.

7.2.4.4. Globalny filtr oddziałów

Stan listy dostępnych oraz aktualnie wybranych oddziałów zarządzany jest za pośrednictwem serwisu wykorzystującego strumienie z biblioteki RxJS. *Observable* reprezentuje strumień wartości emitowanych w czasie, do którego komponenty mogą się podłączyć i reagować na każdą zmianę [134]. *Subject* jest szczególnym rodzajem *Observable,* który umożliwia rozsyłanie wartości do wielu obserwatorów jednocześnie [135]. Spośród dostępnych wariantów wybrano *BehaviorSubject*, ponieważ przechowuje ostatnią wyemitowaną wartość i natychmiast dostarcza ją każdemu nowemu subskrybentowi [136]. Dzięki temu komponenty podłączające się do strumienia po jego inicjalizacji nie tracą aktualnego stanu.

Komponent wyboru oddziału subskrybuje oba strumienie i na ich podstawie renderuje jeden z trzech wariantów widoku, co przedstawiono na *Listing 42*. Oba strumienie zainicjalizowano pustymi tablicami, co umożliwia komponentom natychmiastowe wyrenderowanie. Lista oddziałów zostaje uzupełniona dopiero po zakończeniu inicjalizacji komponentu wyboru departamentów. Serwis pełni rolę jedynego źródła prawdy i udostępnia stan oddziałów pozostałym komponentom wyłącznie do odczytu, co przedstawiono na *Listing 40*. Komponent wyboru oddziału odpowiada za zaznaczanie i odznaczanie wybranych oddziałów przez wywołanie metody zapisu w serwisie, natomiast widoki list odczytują aktualnie wybrane oddziały i stosują je jako filtr przy wysyłaniu zapytań do serwera. Mechanizm ten został omówiony szczegółowo w podpunkcie *7.2.4.5*. Każda operacja modyfikująca listę oddziałów: zapis, edycja i usunięcie po otrzymaniu odpowiedzi serwera propaguje zmianę do wszystkich zasubskrybowanych komponentów, co przedstawiono na *Listing 41.*

Listing 40 Inicjalizacja strumieni oraz metody odczytu i zapisu oddziałów. Źródło: opracowanie własne.

```
private readonly departments$ = new BehaviorSubject<Department[]>([]);
private readonly selectedDepartments$ = new
BehaviorSubject<Department[]>([]);

getDepartments(): Observable<Department[]> {
    return this.departments$.asObservable();
}

setDepartments(departments: Department[]): void {
    this.departments$.next(departments);
}

getSelectedDepartments(): Observable<Department[]> {
    return this.selectedDepartments$.asObservable();
}

setSelectedDepartments(departments: Department[]): void {
```

```
    this.selectedDepartments$.next(departments);
}
```

Listing 41 Propagacja zmiany do BehaviorSubject po operacji zapisu oddziału. Źródło: opracowanie własne.

```
save(department: Department): Observable<Department> {
    return this.http.post<DepartmentJSON>(`${this.url}/create`,
department).pipe(
        map((item) => toDepartment(item)),
        tap((d) => this.departments$.next([...this.departments$.value,
d])),
    );
}
```

Listing 42 Fragmenty kodu szablonu komponentu wyboru oddziałów. Źródło: opracowanie własne.

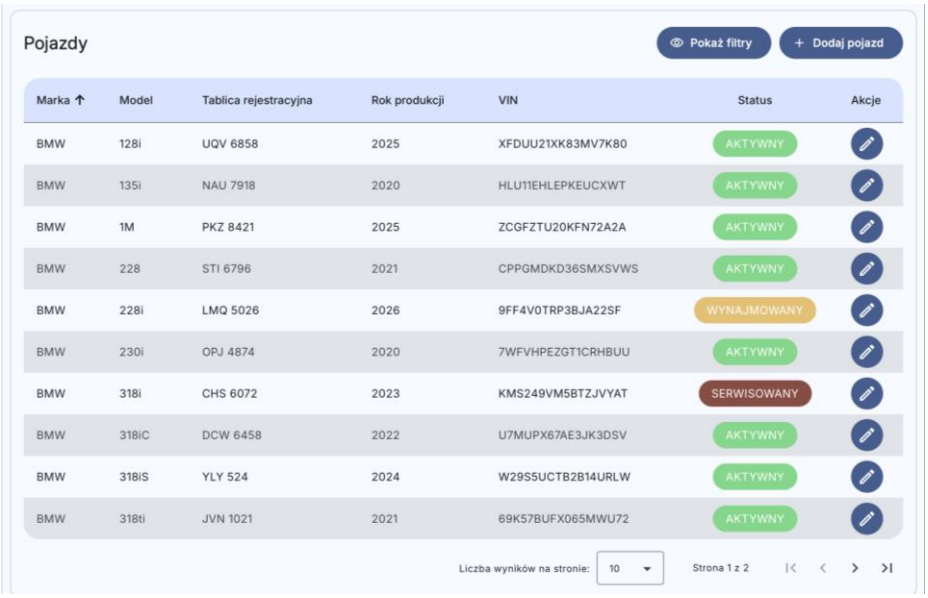
```
@let departments = departments$ | async;
@let selectedDepartments = selectedDepartments$ | async;
@if (departments !== null && selectedDepartments !== null) {
    @if (departments.length === 0) { ... }
    @else if (departments.length === 1) { ... }
    @else {
       <mat-select multiple [value]="selectedDepartments"
                   (selectionChange)="selectDepartments($event)">
          ...
       </mat-select>
    }
}
```

7.2.4.5. Widok listy

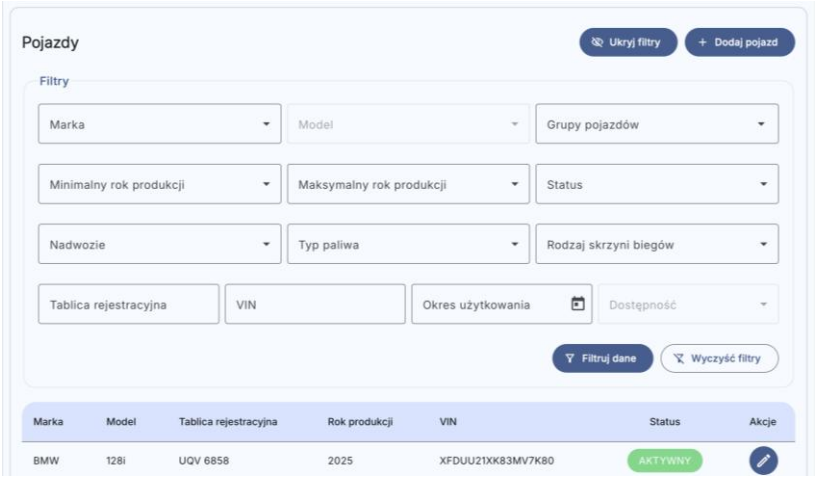
Widok listy zrealizowano bazując na makiecie przedstawionej w podpunkcie *6.4.3.4* oraz komponencie *table* z biblioteki Angular Material. W stosunku do makiety wprowadzono następujące zmiany:

- Kolumny składające się z dwóch pól zostały podzielone, aby umożliwić sortowanie po każdej z nich niezależnie.
- Dane prezentowane w tabeli ograniczono do informacji pozwalających na szybką identyfikację pojazdu.
- Pole wyszukiwania pełnotekstowego usunięto, ponieważ funkcjonalność ta nie została zrealizowana po stronie serwera.
- Filtry przeniesiono do zwijanego panelu opartego na komponencie *expansion panel* z Angular Material, ponieważ liczba filtrów była większa niż przewidywano w makiecie.
- Dodano stronicowanie wyników przy użyciu komponentu *paginator* z Angular Material.

Przykładowa lista została przedstawiona na Rysunek 113, a jej filtry na Rysunek 114.



Rysunek 113 Widok listy pojazdów. Źródło: opracowanie własne.



Rysunek 114 Widok filtrów listy pojazdów. Źródło: opracowanie własne.

Mechanizm pobierania danych z serwera musiał uwzględnić wartości początkowe, którymi były inicjalizowane komponenty służące do: filtrowania, stronicowania i sortowania. Z tego względu został on umieszczony w metodzie cyklu życia `AfterViewInit`, która jest wywoływana dopiero po pełnym zainicjalizowaniu widoku komponentu [137]. Takie rozwiązanie daje pewność, że zapytania `viewChild`, znajdą komponenty przed zbudowaniem strumienia danych.

Strumień zbudowano przy użyciu operatora `combineLatest`, który emituje nową wartość za każdym razem, gdy którekolwiek z połączonych źródeł wyemituje zmianę [138]. Łączy on cztery źródła: strumień filtrów, strumień aktualnie wybranych oddziałów opisany w podpunkcie *7.2.4.4*, zdarzenia zmiany strony oraz zdarzenia zmiany sortowania. Metoda cyklu życia `OnDestroy`jest wywoływana przez Angular w momencie zniszczenia komponentu [50]. Jej wywołanie anuluje subskrypcję strumienia oddziałów, zapobiegając tym samym reagowaniu listy na zmiany filtra oddziałów po opuszczeniu widoku. Zdarzenia stronicowania oraz sortowania zostały uzupełnione operatorem `startWith`, ponieważ domyślnie nie emitują żadnej wartości do czasu pierwszej interakcji użytkownika. Dzięki temu `combineLatest` otrzymuje komplet wartości początkowych od razu po zainicjalizowaniu widoku i może wysłać pierwsze żądanie. Każda zmiana któregokolwiek ze źródeł powoduje wywołanie operatora `switchMap`, który anuluje poprzednie żądanie HTTP i wysyła nowe na podstawie aktualnych wartości filtrów, sortowania i stronicowania [139]. Dlatego też po

zainicjalizowaniu widoku wysyłane jest tylko jedno żądanie. Parametry żądania są konstruowane wewnątrz `switchMap` na podstawie wartości emitowanych przez wszystkie źródła. Powyżej opisany mechanizm został przedstawiony na *Listing 43*.

Wynik żądania przekazywany jest do szablonu za pośrednictwem *async pipe,* który subskrybuje strumień *page$* i przy każdej nowej wartości aktualizuje dane tabeli oraz liczbę rekordów przekazywaną do komponentu *Paginator*, co przedstawiono na *Listing 44*.

Listing 43 Budowanie obiektu żądania pobrania danych do listy pojazdów. Źródło: opracowanie własne.

```
this.page$ = combineLatest([
    this.vehicleFilters$,

this.departmentService.getSelectedDepartments().pipe(takeUntil(this.destr
oy$)),
    this.paginator().page.pipe(startWith({ pageIndex:
this.paginator().pageIndex, pageSize: this.paginator().pageSize })),
    this.sort().sortChange.pipe(startWith({ active: this.sort().active,
direction: this.sort().direction })),
]).pipe(
    switchMap(([filters, departments, page, sort]) => {
       const search = new Search<VehicleFilters>();
       search.filters = filters;
       search.filters.deptIds = departments.map((department) =>
department.id);
       search.filters.accountTypes = this.accountTypes;
       if (sort.direction !== "") {
          search.sort = new Sort();
          search.sort.field = sort.active;
          search.sort.direction = sort.direction;
       }
       search.pagination = new Pagination();
       search.pagination.page = page.pageIndex;
       search.pagination.size = page.pageSize;
       return this.vehicleService.findAll(search);
    }),
);
```

Listing 44 Fragmenty kodu szablonu listy pojazdów. Źródło: opracowanie własne.

```
@let page = (page$ | async) ?? { items: [], totalCount: 0 };
…
<mat-paginator [length]="page.totalCount" [pageSize]="10"
[pageSizeOptions]="[10, 25, 50, 100]"
[showFirstLastButtons]="true"></mat-paginator>
```

Opisany wzorzec jest stosowany we wszystkich widokach list w aplikacji. Poszczególne listy różnią się typem filtrów oraz serwisem dostarczającym dane, natomiast mechanizm reagowania na zmiany filtrów, sortowania i stronicowania pozostaje wspólny.

7.2.4.6. Lista jako alternatywa dla komponentu select

Widok listy opisany w podpunkcie *7.2.4.5* poza trybem przeglądania może służyć do wskazania rekordu w innym formularzu, na przykład pojazdu podczas tworzenia wypożyczenia. Bez tego rozwiązania wskazanie pojazdu wymagałoby osobnego komponentu *select* wraz z własną logiką filtrowania, stronicowania i sortowania. Aby tego uniknąć, przyjęto, że rolę tę przejmuje gotowy komponent listy uruchamiany w trybie wyboru. W ten sposób wykorzystywany jest istniejący mechanizm bez potrzeby budowania go ponownie dla pojedynczego pola. Dodatkowym powodem rezygnacji z listy rozwijanej była trudność z czytelnym przedstawieniem rekordów złożonych z kilku pól. W komponencie select każdą pozycję trzeba sprowadzić do pojedynczej etykiety, co utrudnia

rozróżnienie podobnych rekordów. Tabela pozwala rozłożyć te same dane na osobne kolumny, dzięki czemu rozpoznanie i wybór właściwego rekordu są prostsze. O trybie wyboru decyduje obecność konfiguracji przekazanej jako dane okna dialogowego, której strukturę przedstawia *Listing 45*. Jeśli konfiguracja zostanie wstrzyknięta, lista działa w trybie wyboru. W przeciwnym razie zachowuje się jak zwykły widok przeglądania. Dzięki temu rozróżnieniu jeden komponent obsługuje oba zastosowania. Dalsze zachowanie wynika z wartości odczytanych z konfiguracji, co przedstawia *Listing 46*.

Listing 45. Interfejs listy w trybie wyboru. Źródło: opracowanie własne.

```
export interface SelectableListConfig<T, F = unknown> {
    multiple: boolean;
    initiallySelectedValues?: T[];
    initialFilters?: F;
    filtersOpenedInitially?: boolean;
    disabledFiltersNames?: string[];
}
```

Listing 46 Fragmenty kodu komponentu listy pojazdów. Źródło: opracowanie własne.

```
readonly dialogRef = inject<MatDialogRef<VehicleListComponent, Vehicle[]
| Vehicle> | null>(MatDialogRef<VehicleListComponent>, { optional: true
});
readonly config = inject<SelectableListConfig<Vehicle, VehicleFilters> |
null>(MAT_DIALOG_DATA, { optional: true });
…
readonly listSelectable = computed(() => this.config !== null);
readonly selectionMultiple = computed(() => this.config?.multiple);
readonly initiallySelectedVehicles = computed(() =>
this.config?.initiallySelectedValues);
readonly initiallyFiltered = computed(() => this.config?.initialFilters
!== undefined);
readonly initialFilters = computed(() => this.config?.initialFilters);
readonly disabledFiltersNames = computed(() =>
this.config?.disabledFiltersNames ?? []);
readonly filtersOpenedInitially = computed(() =>
this.config?.filtersOpenedInitially ?? false);
…
readonly columns = ["make", "model", "licensePlate", "yearOfProduction",
"vin", "status"];
readonly displayedColumns = computed(() => (this.listSelectable() ?
["select", ...this.columns] : [...this.columns, "actions"]));
readonly selection = new MultipleSelection<Vehicle>((a, b) => a.id ===
b.id, this.initiallySelectedVehicles());
```

Zaznaczenie obsługuje klasa rozszerzająca `SelectionModel` z Angular Component Dev Kit (CDK), co przedstawia *Listing 47*. Rekordy porównywane są po identyfikatorze, co zapewnia poprawne odtworzenie wstępnie wskazanych pozycji niezależnie od ich kolejności na liście. Wynik zwracany jest przez zamknięcie okna dialogowego. W trybie pojedynczym następuje to natychmiast po wskazaniu rekordu, a w trybie wielokrotnym dopiero po potwierdzeniu zaznaczenia.

Listing 47 Klasa MultipleSelection rozszerzająca SelectionModel. Źródło: opracowanie własne.

```
export class MultipleSelection<T> extends SelectionModel<T> {
    constructor(compareWith: (o1: T, o2: T) => boolean,
initiallySelectedValues?: T[]) {
        super(true, initiallySelectedValues, true, compareWith);
    }

    isAllSelected(values: T[]): boolean {
        return values.length > 0 && values.every((value) =>
this.isSelected(value));
    }
```

```
    toggleAll(values: T[]): void {
       if (this.isAllSelected(values)) {
          this.deselect(...values);
       } else {
          this.select(...values);
       }
    }
}
```

Zestaw kolumn wyznaczany jest dynamicznie: w trybie wyboru kolumnę akcji zastępuje kolumna zaznaczania. Sposób zaznaczania zależy od trybu. Przy wyborze wielokrotnym stosowany jest komponent *checkbox* z Angular Material wraz z opcją zaznaczenia wszystkich rekordów bieżącej strony, a przy wyborze pojedynczym komponent *radio button* z Angular Material. W trybie wyboru zablokowane jest też wejście w szczegóły rekordu oraz akcje dostępne zwykle w prawym górnym rogu, a lista ma inny tytuł. Tryb wyboru wpływa również na stopkę tabeli. Przy wyborze wielokrotnym pojawiają się przyciski potwierdzenia i wyczyszczenia zaznaczenia wraz z liczbą wybranych rekordów, a przy pojedynczym przycisk anulowania. Opisane fragmenty szablonu przedstawia *Listing 48*.

Listing 48 Fragmenty kodu szablonu komponentu listy pojazdów. Źródło: opracowanie własne.

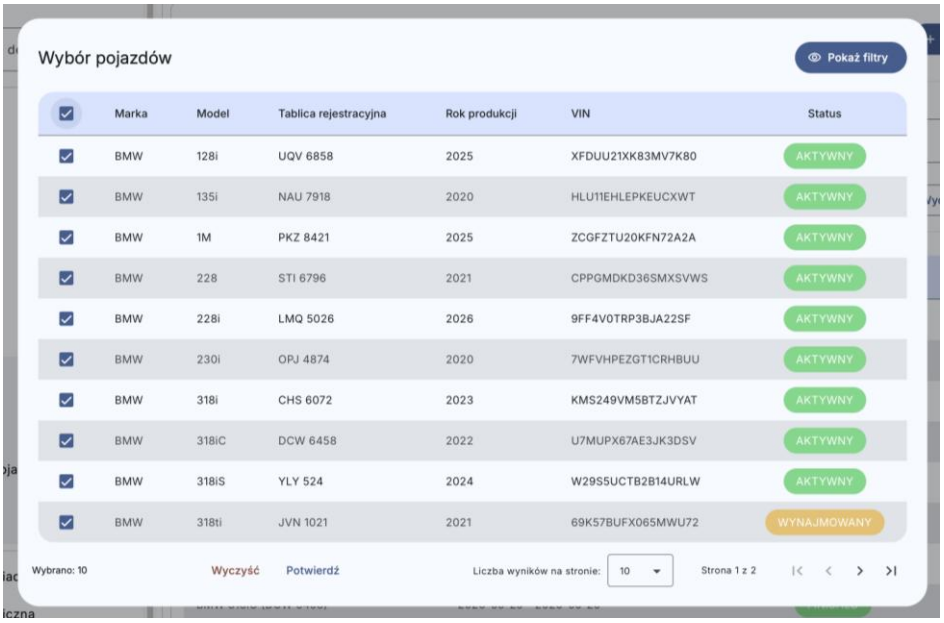
```
@if (listSelectable()) {
    <ng-container matColumnDef="select">
       <th mat-header-cell *matHeaderCellDef>
          @if (selectionMultiple()!) {
             <mat-checkbox (change)="selection.toggleAll(page.items)"
[checked]="selection.isAllSelected(page.items)" />
          }
       </th>
       <td mat-cell *matCellDef="let vehicle">
          @if (selectionMultiple()!) {
             <mat-checkbox (change)="selection.toggle(vehicle)"
[checked]="selection.isSelected(vehicle)" />
          } @else {
             <mat-radio-button (click)="dialogRef?.close(vehicle)"
[checked]="selection.isSelected(vehicle)" />
          }
       </td>
    </ng-container>
}
```

W formularzu lista osadzona jest jako pole przypominające komponent *select* z Angular Material, co przedstawia *Listing 49*.

Listing 49 Pole wyboru pojazdu w trybie listy. Źródło: opracowanie własne.

```
<mat-form-field appearance="outline">
    <mat-label>{{ "vehicles" | translate }}</mat-label>
    <input matInput readonly [value]="vehiclesControl?.value?.length ?
('selected' | translate: { number: vehiclesControl!.value!.length }) :
null"  (click)="selectVehicle()">
    <mat-icon matSuffix>arrow_drop_down</mat-icon>
    @if (vehiclesControl?.value !== null) {
       <button type="button" matIconButton matSuffix
             (click)="vehiclesControl?.reset(); $event.preventDefault()">
          <mat-icon>clear</mat-icon>
       </button>
    }
</mat-form-field>
```

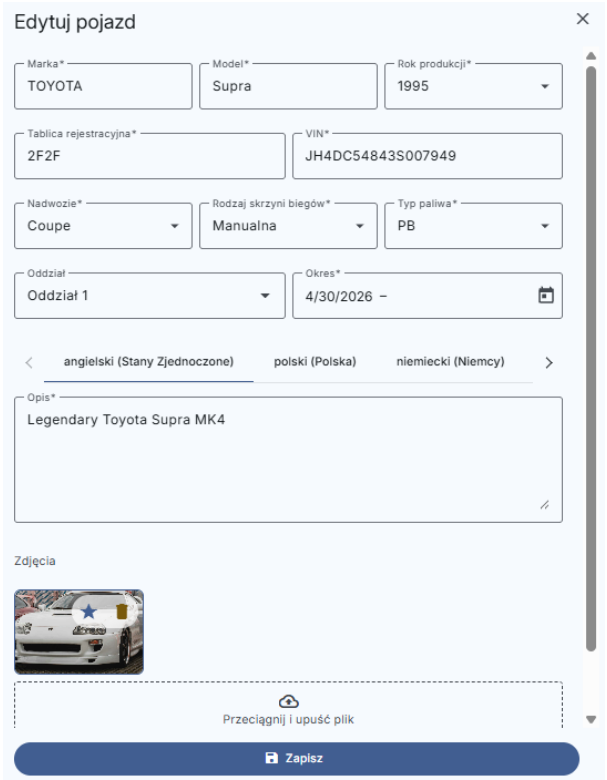
Okno listy otwartej w trybie wyboru przedstawia Rysunek 115.



Rysunek 115 Okno listy pojazdów w trybie wyboru. Źródło: opracowanie własne.

7.2.4.7. Okno dodawania i edycji

Okno dodawania oraz edycji zrealizowano z wykorzystaniem wspólnego komponentu opartego na *dialog* z Angular Material, co pozwoliło uniknąć powielania kodu. Mechanizm ten jest wspólny dla wszystkich modułów systemu. Został on przedstawiony na przykładzie widoku pojazdów. Widok został oparty na makietach przedstawionych w punkcie *6.4.3.6*, jednak w toku realizacji projektu zakres formularza uległ rozszerzeniu. W końcowej implementacji zrezygnowano z wielokrokowego układu na rzecz pojedynczego formularza. Poza polami uwzględnionymi w makiecie, formularz uzupełniono o pole przypisania pojazdu do oddziału zgodnie z wymaganiami. Największą zmianą względem makiety było dodanie opcjonalnej sekcji pierwszego stanu technicznego pojazdu przy dodawaniu. *Rysunek 116* przedstawia opisany formularz dodania nowego pojazdu.



Rysunek 116 Okno edycji pojazdu. Źródło: opracowanie własne.

Tłumaczenia opisów na wiele języków

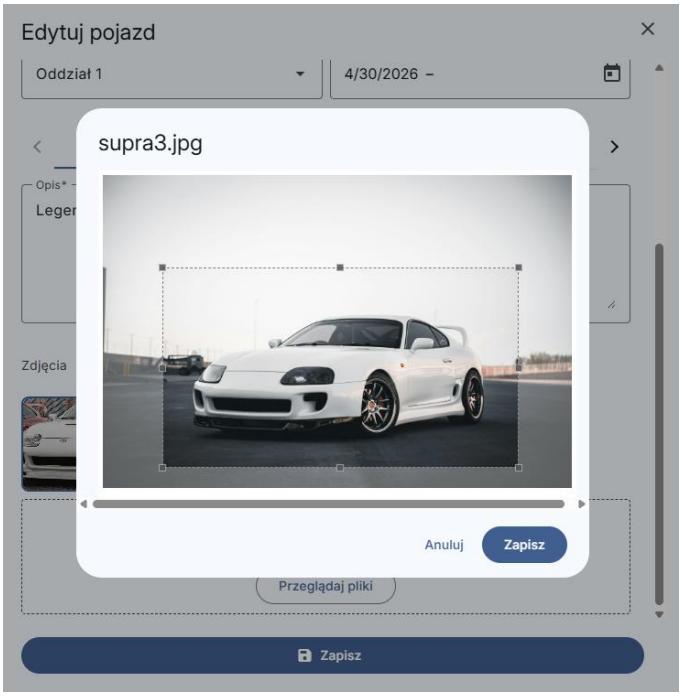
Opis pojazdu może być wprowadzany w wielu językach, zgodnie z językami obsługiwanymi przez wypożyczalnię. Do obsługi tłumaczeń zastosowano pomocniczą funkcję `formRecord`, która tworzy grupę kontrolek formularza z jedną kontrolką dla każdego języka. Funkcję tę przedstawiono na *Listing 50*.

Listing 50 Pomocnicza funkcja formRecord tworząca grupę kontrolek opisu dla obsługiwanych przez wypożyczalnię języków. Źródło: opracowanie własne.

```
export function formRecord<V>
(record: Record<string, V>,
languages: string[],
primaryLanguage: string
): FormRecord<FormControl<V | null>> {
    const formRecord = new FormRecord<FormControl<V | null>>({});
    for (const language of languages) {
       formRecord.addControl(language, new FormControl(record[language],
language === primaryLanguage ? Validators.required : undefined));
    }
    return formRecord;
}
```

Przycinanie zdjęć

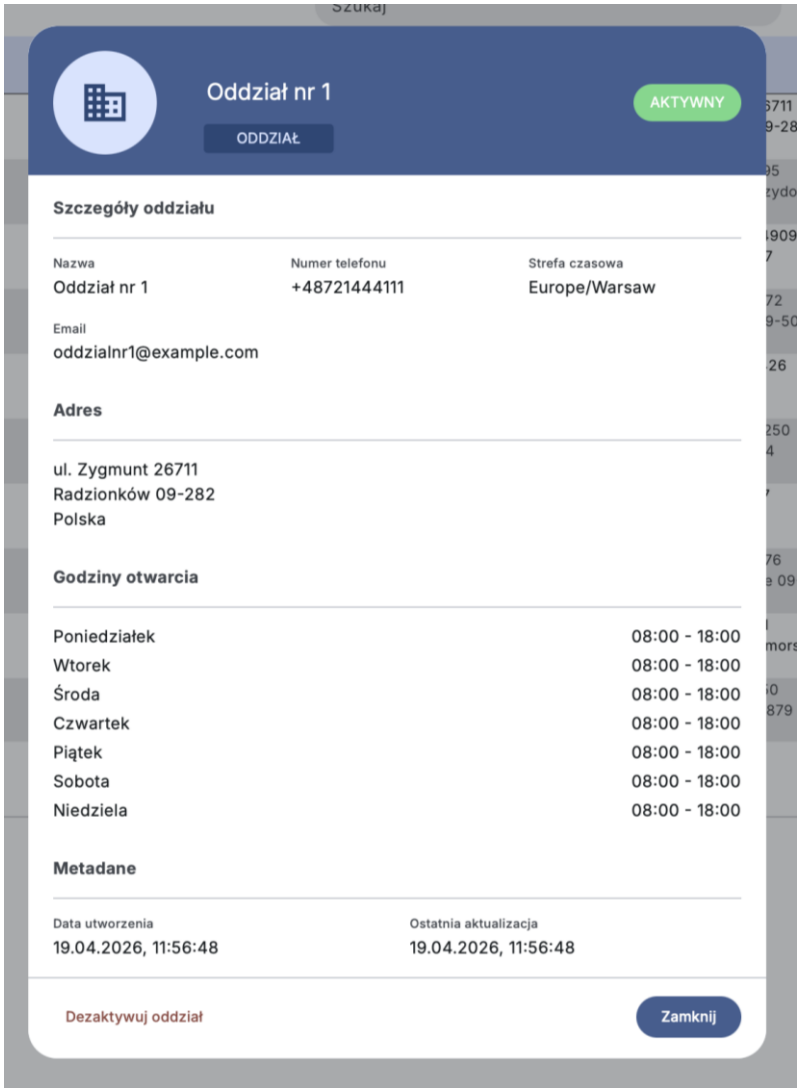
Zdjęcia pojazdu dodawane są przez obszar przeciągania plików lub wybór plików z dysku. Do przycinania zdjęć przed ich dołączeniem do formularza zastosowano bibliotekę *ngx-image-cropper*. Dzięki niej zdjęcia trafiające do systemu mają ujednolicone proporcje, co ułatwia ich późniejszą prezentację w widokach ofert na stronie wypożyczalni. *Rysunek 117* przedstawia okno dialogowe kadrowania zdjęć, otwierane po wybraniu pliku przez użytkownika.



Rysunek 117 Widok komponentu kadrowania zdjęć w formularzu dodawania/edycji pojazdu. Źródło: opracowanie własne.

7.2.4.8. Okno szczegółów

Okno szczegółów zaimplementowano przy użyciu komponentu *dialog* z Angular Material, bazując na makiecie przedstawionej w podpunkcie *6.4.3.5. Rysunek 118* przedstawia przykładowe okno szczegółów oddziału, lecz sposób działania tego komponentu pozostaje jednakowy dla wszystkich obiektów w systemie.



Rysunek 118 Okno szczegółów oddziału. Źródło: opracowanie własne.

7.2.4.9. Dodawanie szablonów wiadomości

Formularz dodawania szablonu wiadomości zaprezentowany na *Rysunek 119* został oparty na standardowych komponentach z Angular Material poza obsługą treści sformatowanej oraz kodu HTML.



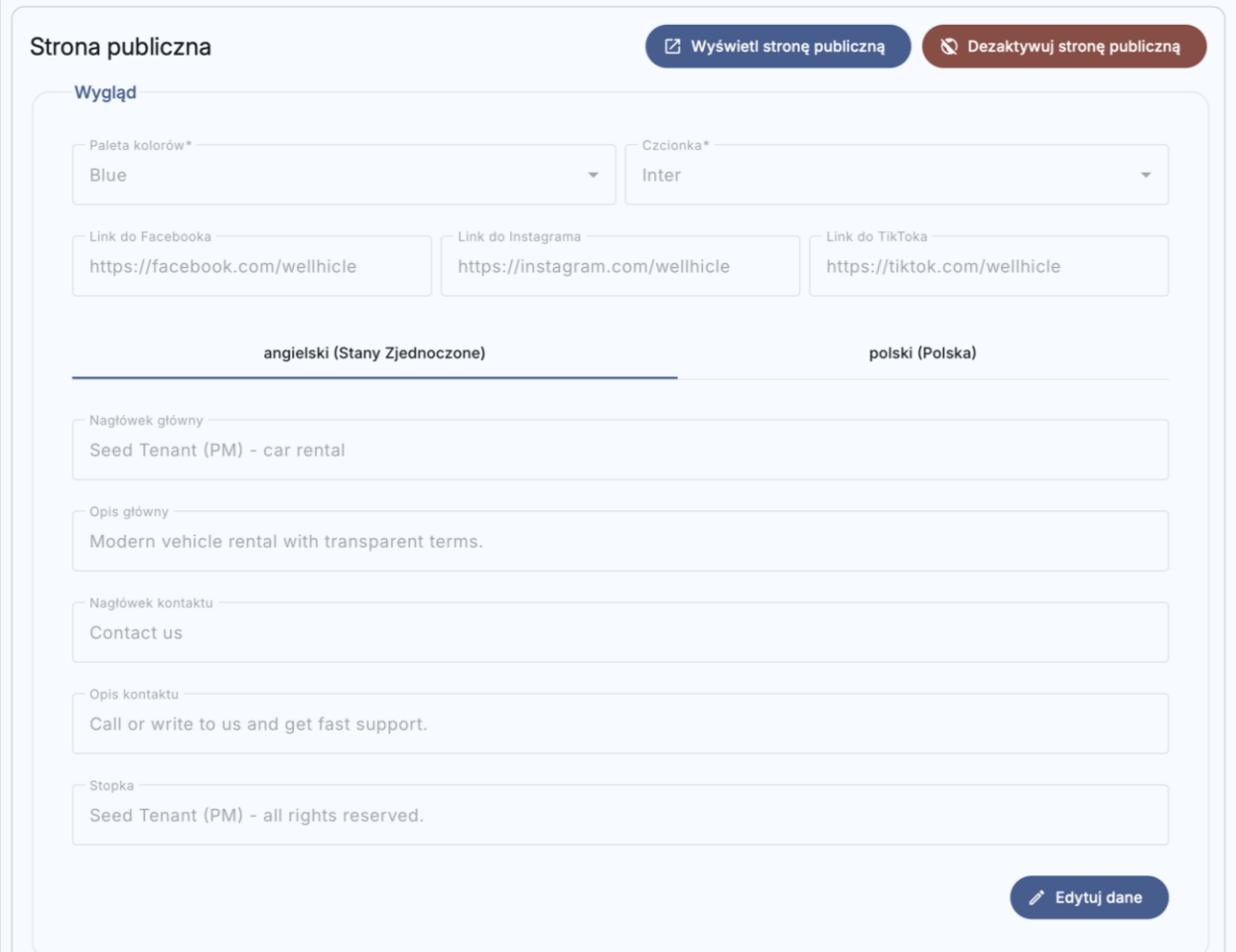
Rysunek 119 Okno formularza tworzenia szablonu wiadomości. Źródło: opracowanie własne.

Edycja treści wiadomości została zrealizowana z użyciem biblioteki ngx-editor, która udostępnia do tego celu komponent zbudowany w Angularze. Dodatkowo dodano możliwość edycji kodu HTML przy użyciu biblioteki @fsegurai/ngx-codemirror, który udostępnia komponent zbudowany w Angularze oparty na edytorze CodeMirror. Ze względu na możliwość ręcznej edycji kodu HTML przed wysłaniem treści do serwera zastosowano bibliotekę DOMPurify. DOMPurify służy do oczyszczania m.in. kodu HTML oraz ograniczania ryzyka ataków XSS przez usuwanie potencjalnie niebezpiecznych elementów [140]. W komponencie biblioteka DOMPurify została wykorzystana przed wysłaniem formularza. Treść pola jest oczyszczana metodą `sanitize`, a dopiero następnie przypisywana do modelu przekazywanego do serwera.

7.2.4.10. Widok strony publicznej

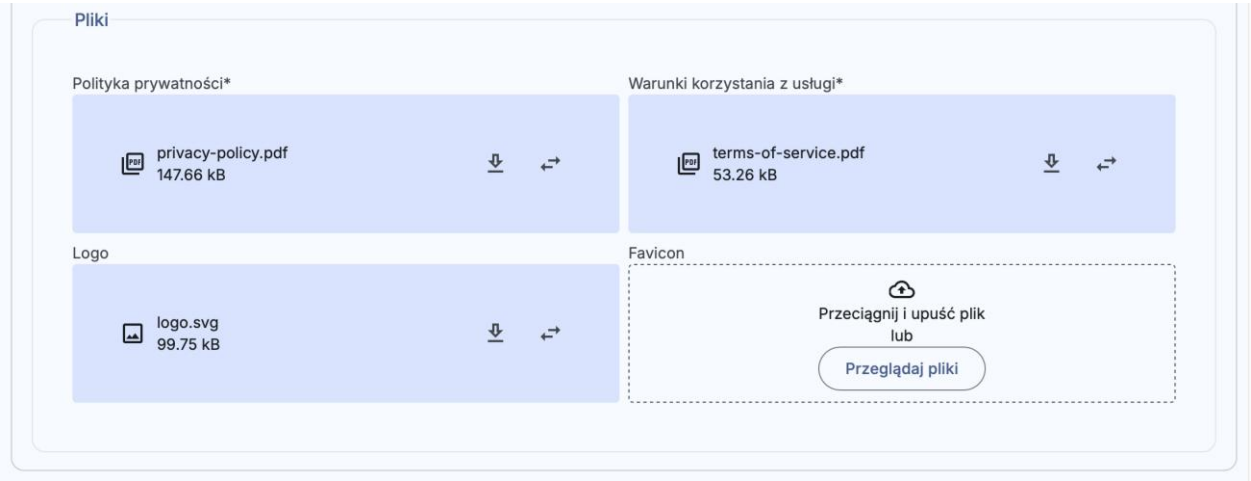
Widok strony publicznej nie został objęty projektem interfejsu przez zespół *designu*, dlatego opracowano go od podstaw na etapie implementacji. Jako punkt odniesienia przyjęto osadzenie widoku w komponencie *card* w celu zachowania spójności z widokiem listy oraz podział na sekcje przy użyciu elementu *fieldset,* co przedstawiono na *Rysunek 120*. Samodzielnie przygotowano również obszar przeciągania plików, co przedstawiono na *Rysunek* 121. Aktywacja strony publicznej, dostępna pod przyciskiem w prawym górnym rogu widoku, możliwa jest dopiero po wgraniu wymaganych plików. Po aktywacji w tym samym miejscu udostępniany jest przycisk dezaktywacji, umożliwiający odwrócenie tej akcji. Obie akcje wymagają potwierdzenia, do czego wykorzystano komponent opisany w podpunkcie *7.2.2.1*.

Formularz wyglądu domyślnie ładowany jest w stanie wyłączonym, a jego modyfikacja wymaga jawnego przejścia w tryb edycji. Przed rozpoczęciem edycji zapisywana jest kopia wartości formularza. Anulowanie przywraca tę kopię i ponownie blokuje formularz, dzięki czemu rezygnacja ze zmian nie pozostawia w polach wartości częściowo zmodyfikowanych.



Rysunek 120 Sekcja wyglądu strony publicznej. Źródło: opracowanie własne.

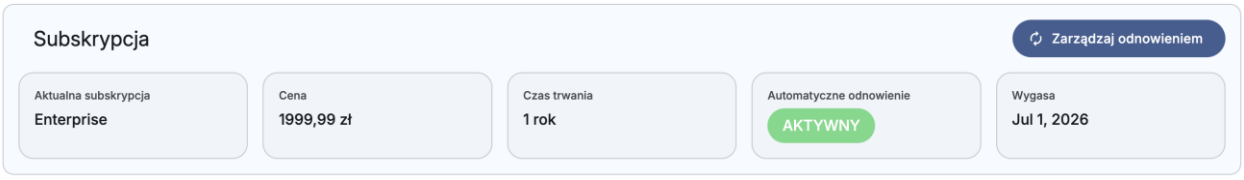
Plików nie można usuwać, a jedynie podmieniać. Z tego względu udostępniono przycisk podmiany, który usuwa z pola dotychczasowy plik i przywraca obszar wczytywania, umożliwiając załadowanie nowego pliku w jego miejsce.



Rysunek 121 Sekcja plików strony publicznej. Źródło: opracowanie własne.

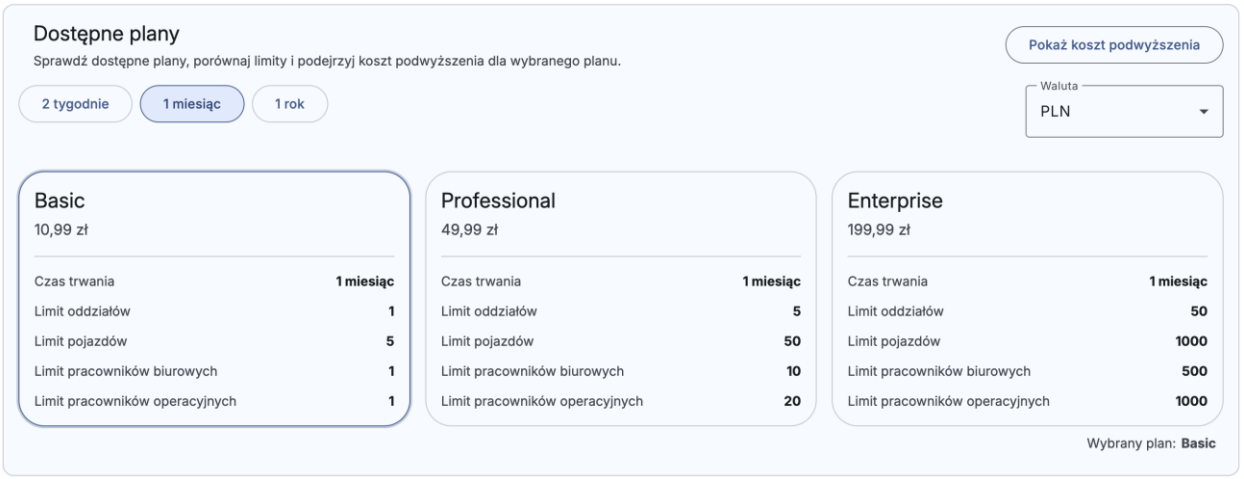
7.2.4.11. Widok subskrypcji

Widok subskrypcji dostępny jest tylko dla właściciela. Główna część widoku została podzielona na trzy sekcje. Pierwsza z nich prezentuje aktualną subskrypcję, co przedstawiono na *Rysunek 122*. W sekcji udostępniono również zarządzanie automatycznym odnowieniem.



Rysunek 122 Widok aktualnej subskrypcji. Źródło: opracowanie własne.

Druga sekcja obejmuje dostępne plany subskrypcji. Plany prezentowane są w formie kart bazujących na komponencie *card* z Angular Material. Dodatkowo udostępniono filtry okresu trwania oraz waluty, co przedstawiono na *Rysunek 123*.



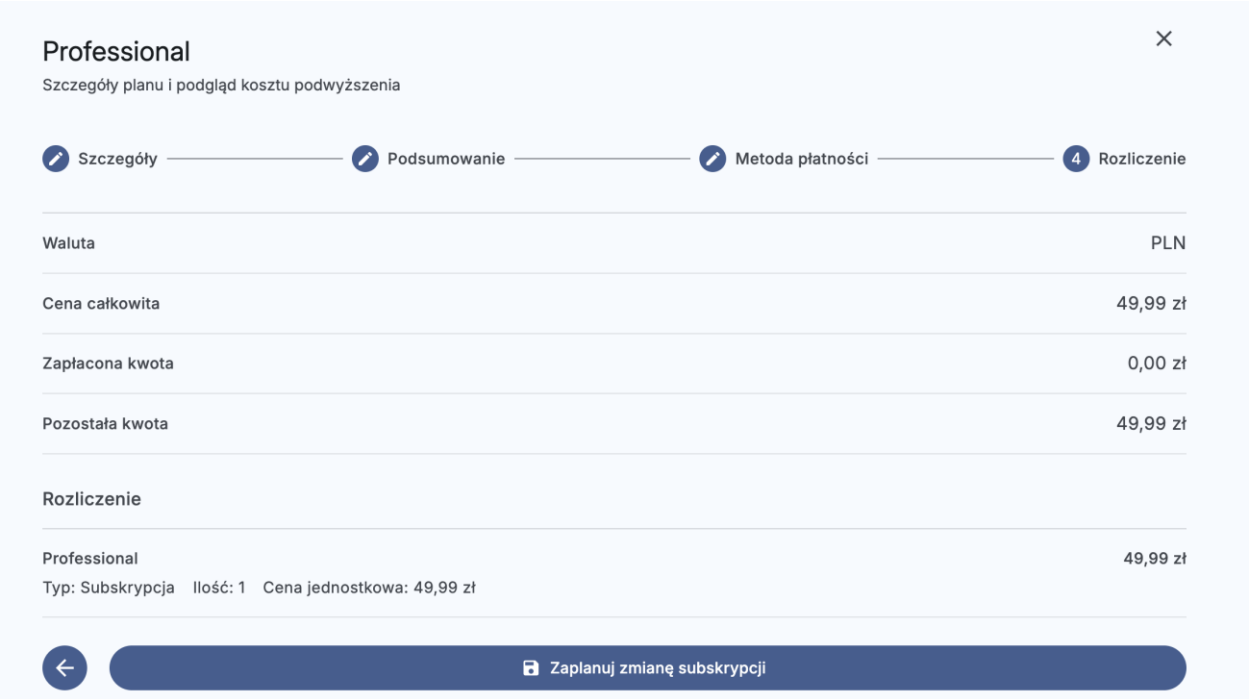
Rysunek 123 Widok dostępnych planów subskrypcji. Źródło: opracowanie własne.

Zmiana planu odbywa się w oknie opartym na komponencie *stepper* z Angular Material. Pierwszy krok porównuje bieżący oraz wybrany plan. Kolejny krok pokazuje, czy nastąpiło podwyższenie, czy obniżenie kosztu i od kiedy zmiana ma zastosowanie. W przypadku podwyższenia planu system pobiera z serwera koszt podwyższenia, natomiast w przypadku obniżenia planu zmiana zostaje wdrożona po zakończeniu bieżącego okresu subskrypcji. Po wybraniu subskrypcji aplikacja pobiera podgląd rozliczenia.



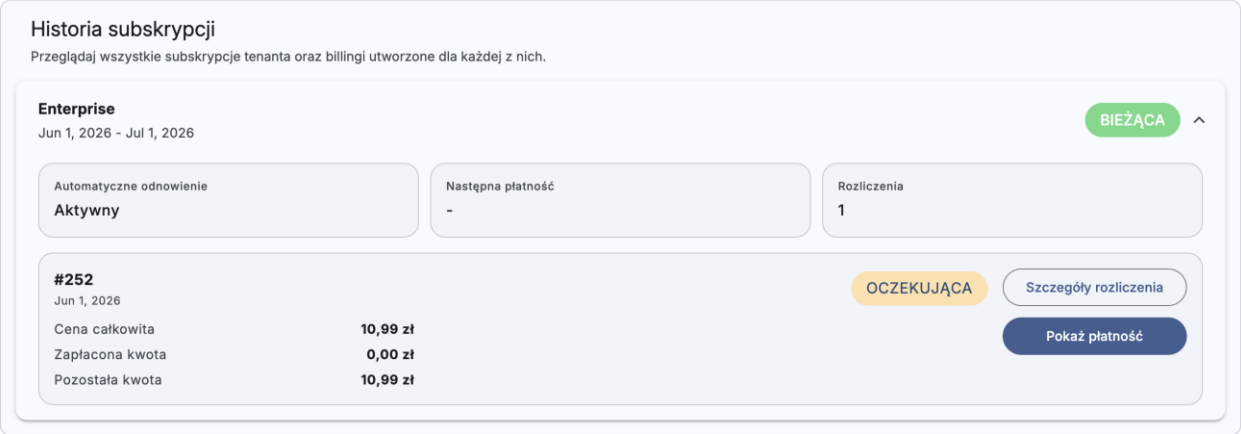
Rysunek 124 Widok kreatora zmiany planu subskrypcji – krok 1. Źródło: opracowanie własne.

Podgląd ten jest prezentowany przed zatwierdzeniem operacji. Po zapisaniu zmiany aplikacja w razie potrzeby inicjuje wymaganą płatność. Na *Rysunek 125* przedstawiono widok poglądu rozliczenia podwyższenia subskrypcji.



Rysunek 125 Widok podglądu rozliczenia zmiany subskrypcji. Źródło: opracowanie własne.

Ostatnia sekcja widoku prezentuje historię subskrypcji wraz z powiązanymi rozliczeniami. Dzięki temu moduł subskrypcji łączy informacje o planie z modułem finansowym, bez konieczności ręcznego wyszukiwania powiązanych dokumentów w osobnych widokach. Na *Rysunek 126* przedstawiono sekcję historii subskrypcji oraz szczegóły pojedynczej subskrypcji klienta platformy.

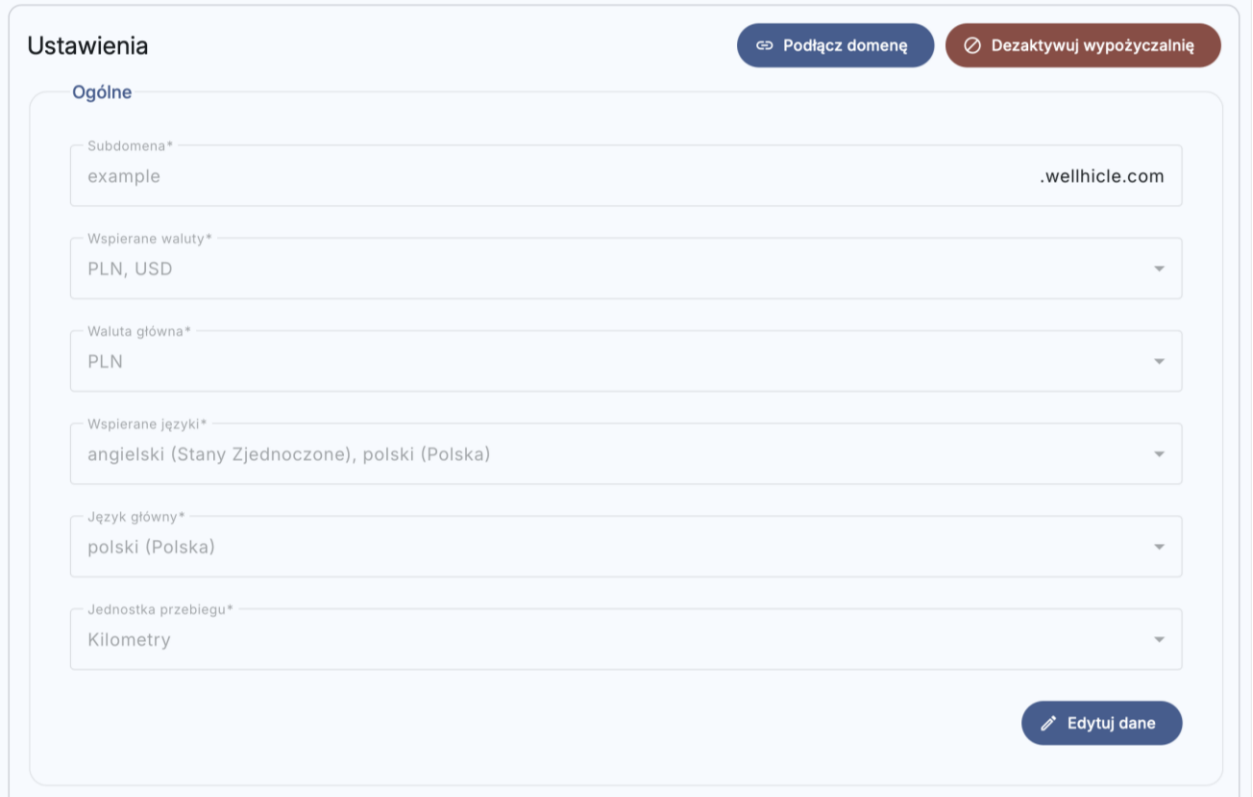


Rysunek 126 Widok historii subskrypcji. Źródło: opracowanie własne.

7.2.4.12. Widok ustawień

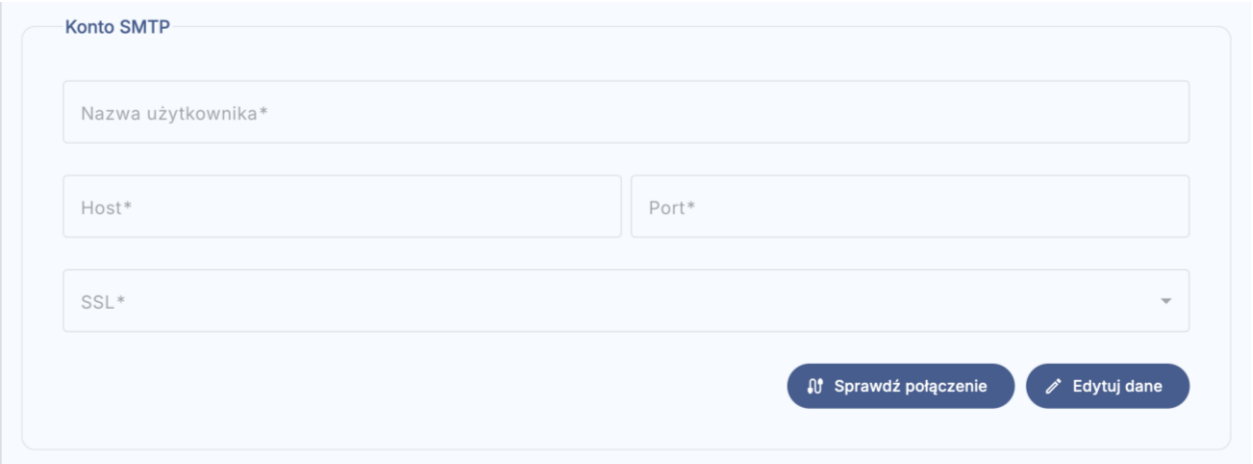
Widok ustawień bazuje na widoku strony publicznej opisanym w podpunkcie *7.2.4.10*. Podobnie jak tam, konfiguracja wypożyczalni została pogrupowana w sekcje odpowiadające obszarom funkcjonalnym, co ogranicza liczbę informacji prezentowanych jednocześnie. Dezaktywacja wypożyczalni dostępna jest pod przyciskiem w prawym górnym rogu widoku i wymaga potwierdzenia, do czego wykorzystano komponent opisany w podpunkcie *7.2.2.1*. Po jej wykonaniu następuje przekierowanie do ekranu logowania, z poziomu którego możliwa jest późniejsza reaktywacja, opisana w podpunkcie *7.2.4.14*. Konfigurację domeny dostępną pod przyciskiem „Podłącz domenę “, ze względu na obszerność wynikającą z czterokrokowego formularza, wydzielono do podpunktu *7.2.4.13*.

Sekcja ustawień ogólnych przedstawiona na *Rysunek 127* prezentuje konfigurację wypożyczalni o zasięgu ogólnosystemowym. Wartości te są ustalane w trzecim kroku kreatora wypożyczalni, opisanym w podpunkcie *7.2.4.1*, a w omawianym widoku pozostają dostępne do podglądu i edycji.



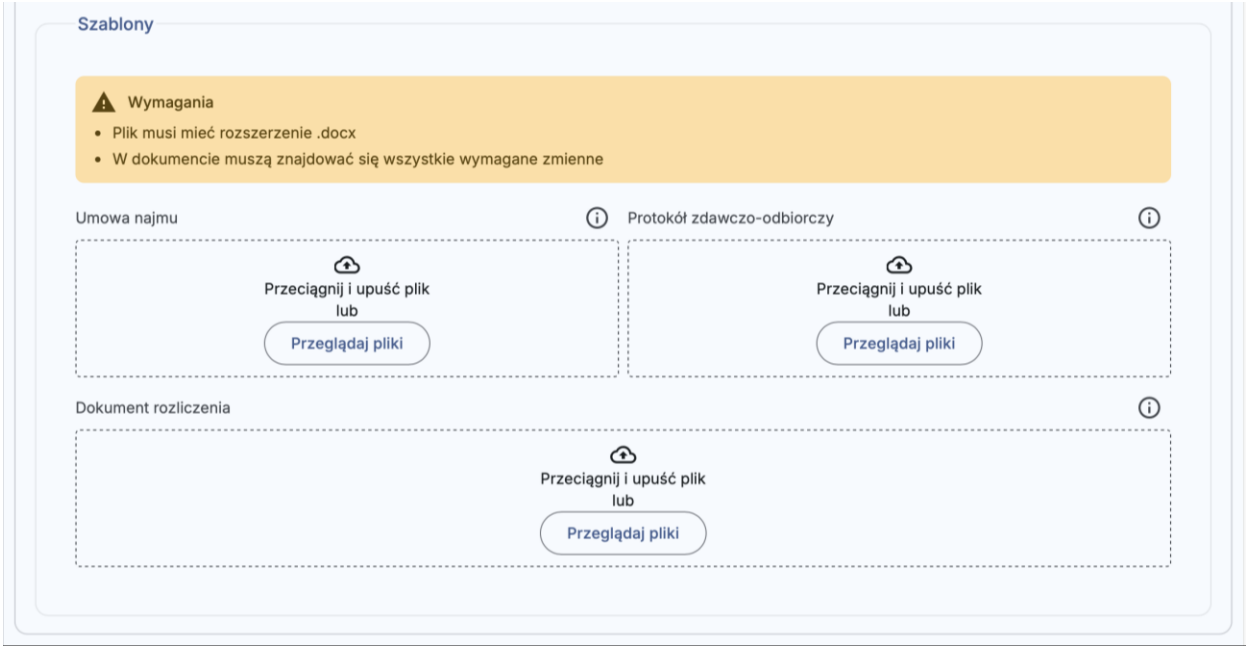
Rysunek 127 Sekcja ustawień ogólnych. Źródło: opracowanie własne.

Sekcja konta pocztowego przedstawiona na *Rysunek 128* obejmuje konfigurację serwera SMTP, z którego wysyłane są wiadomości w imieniu wypożyczalni. Dodatkowo została udostępniona możliwość sprawdzenia połączenia w celu weryfikacji wprowadzonych danych.



Rysunek 128 Sekcja ustawień konta SMTP. Źródło: opracowanie własne.

Sekcja szablonów dokumentów przedstawiona na *Rysunek 129* z perspektywy aplikacji przeglądarkowej ogranicza się do przyjęcia pliku o rozszerzeniu .docx, analogicznie do mechanizmu plików opisanego w podpunkcie *7.2.4.10*. Dodatkowo po najechaniu na ikonę informacji prezentowana jest lista wszystkich dostępnych zmiennych wraz z możliwością ich skopiowania, co ułatwia przygotowanie poprawnego szablonu.

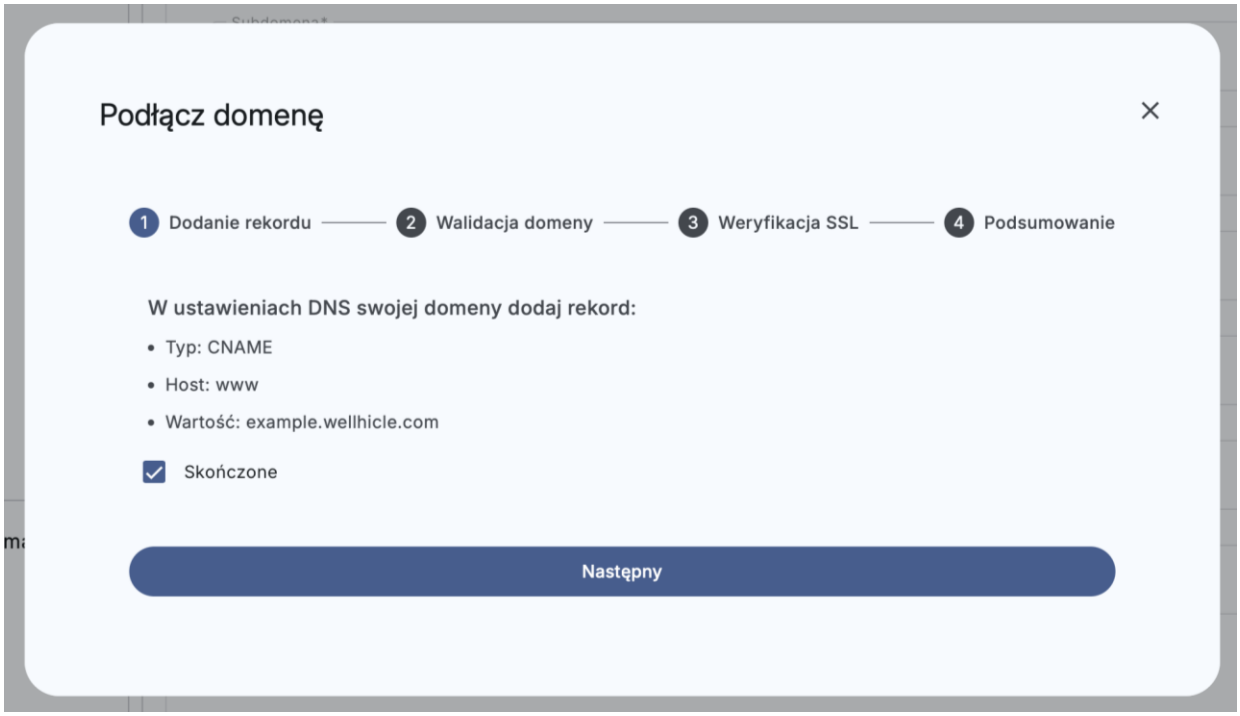


Rysunek 129 Sekcja ustawień szablonów. Źródło: opracowanie własne.

7.2.4.13. Podłączanie domeny

Podłączanie domeny dostępne jest z poziomu ustawień, które opisano w podpunkcie *7.2.4.12*. Widok nie posiadał makiety, dlatego jego układ wzorowano na kreatorze wypożyczalni, opisanym w podpunkcie *7.2.4.1*. Zrealizowano go w oknie dialogowym jako formularz czterokrokowy z wykorzystaniem komponentu *stepper* z Angular Material.

Pierwszy krok, przedstawiony na *Rysunek 130*, zawiera instrukcję dodania rekordu CNAME u dostawcy domeny. Przejście do kolejnego kroku wymaga zaznaczenia pola potwierdzenia wykonania wcześniejszej czynności w celu zwrócenia uwagi użytkownika.



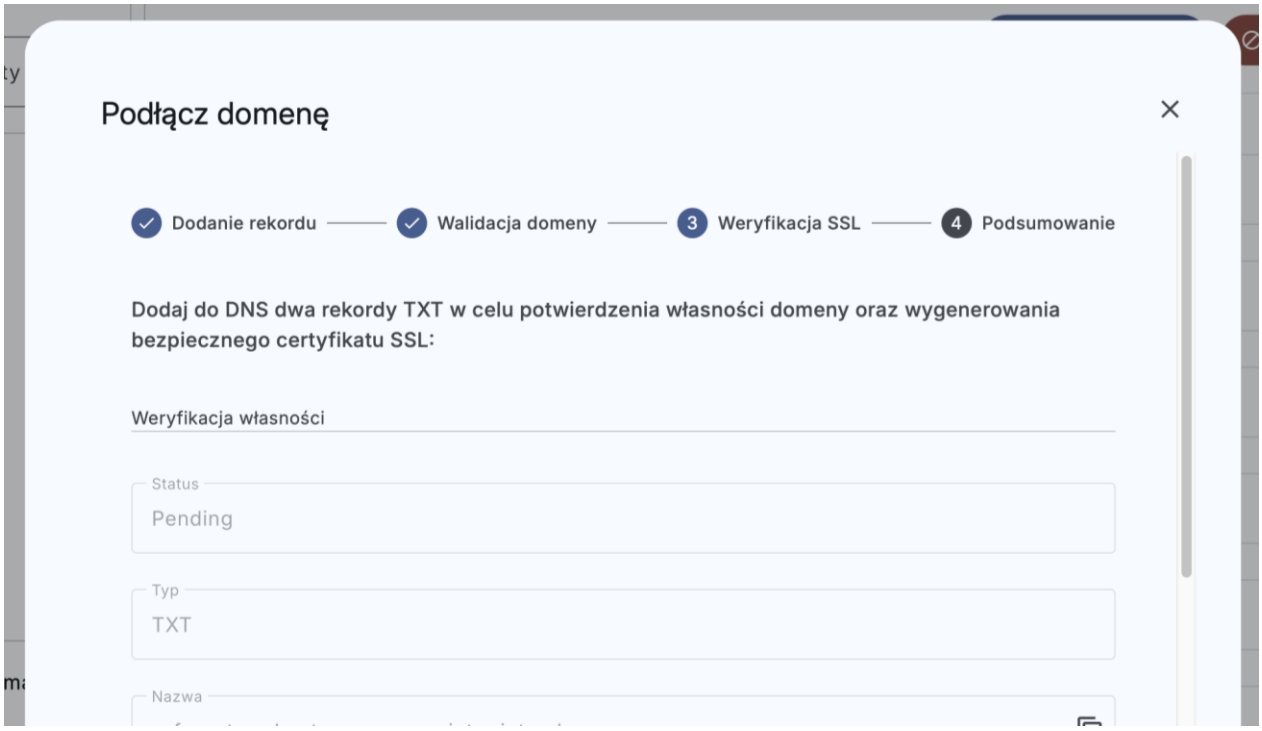
Rysunek 130 Kreator podłączania domeny - krok 1. Źródło: opracowanie własne.

Drugi krok, przedstawiony na *Rysunek 131*, obejmuje podanie domeny. Walidacja pola łączy reguły synchroniczne z asynchroniczną weryfikacją po stronie serwera, analogicznie do pola subdomeny w trzecim kroku kreatora wypożyczalni, opisanym w podpunkcie *7.2.4.1*.



Rysunek 131 Kreator podłączania domeny - krok 2. Źródło: opracowanie własne.

Trzeci krok, przedstawiony na *Rysunek 132*, prezentuje rekordy DNS zwrócone przez dostawcę. Rekordy wyświetlane są w polach tylko do odczytu, a przy ich nazwie oraz wartości udostępniono przyciski kopiowania do schowka, co ułatwia przeniesienie tych danych.



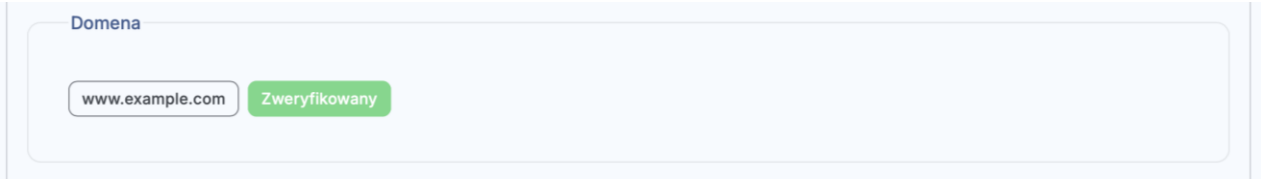
Rysunek 132 Kreator podłączania domeny - krok 3. Źródło: opracowanie własne.

Czwarty krok, przedstawiony na *Rysunek 133*, stanowi ekran podsumowujący. Po podaniu domeny i wygenerowaniu rekordów powrót do wcześniejszych kroków jest zablokowany, co zabezpiecza przed ponowną konfiguracją prowadzącą do niespójnego stanu.



Rysunek 133 Kreator podłączania domeny - krok 4. Źródło: opracowanie własne.

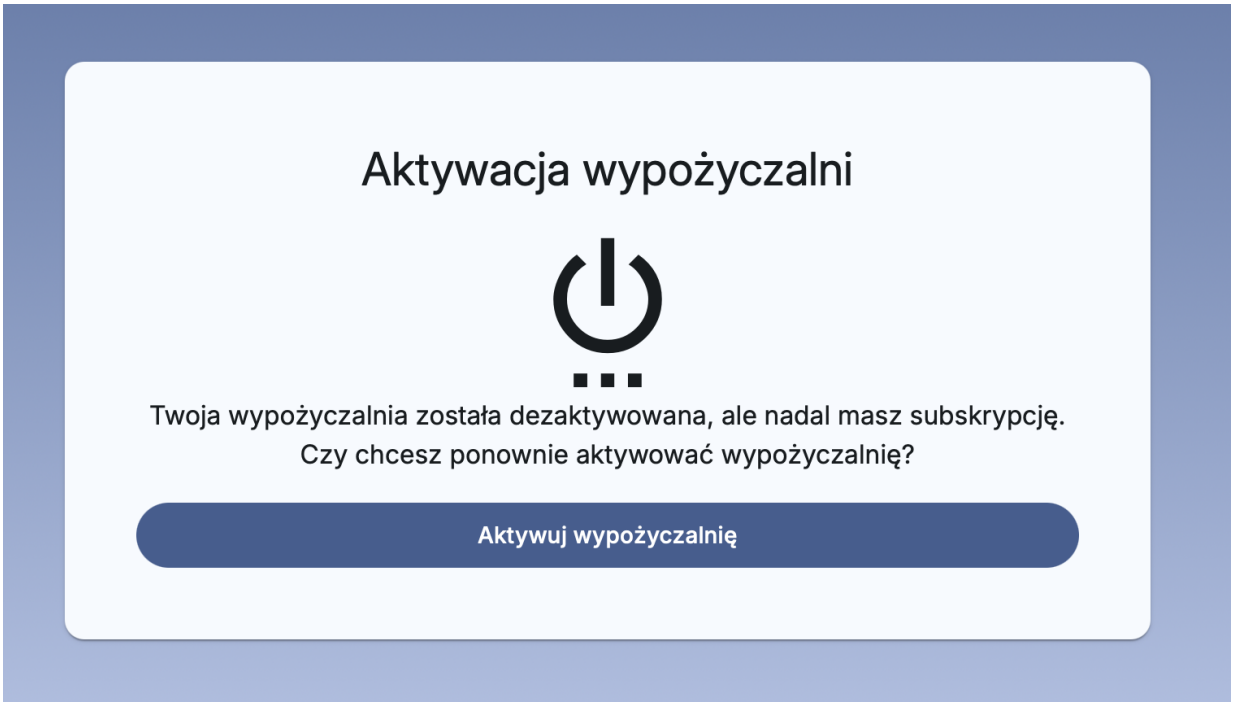
Po pomyślnym podłączeniu domeny w widoku ustawień pojawia się poświęcona jej sekcja, co przedstawiono na *Rysunek 134*. Dopóki domena pozostaje niezweryfikowana, w sekcji tej dostępny jest przycisk sprawdzający oraz aktualizujący jej status.



Rysunek 134 Sekcja ustawień domeny. Źródło: opracowanie własne.

7.2.4.14. Reaktywacja wypożyczalni

Reaktywacja rozpoczyna się na ekranie logowania, do którego właściciel zostaje przekierowany po dezaktywacji. Próba zalogowania do dezaktywowanej wypożyczalni uruchamia dalszy przepływ reaktywacji dostępny wyłącznie dla właściciela. O dalszej ścieżce decyduje stan bieżącej subskrypcji. Jeśli subskrypcja pozostaje aktywna, właściciel trafia do widoku aktywacji wypożyczalni, przedstawionego na *Rysunek 135*. Jeśli subskrypcja wygasła, kierowany jest do widoku odnowienia subskrypcji, przedstawionego na *Rysunek 136*, którego przycisk prowadzi do kreatora subskrypcji.



Rysunek 135. Widok aktywacji wypożyczalni. Źródło: opracowanie własne.

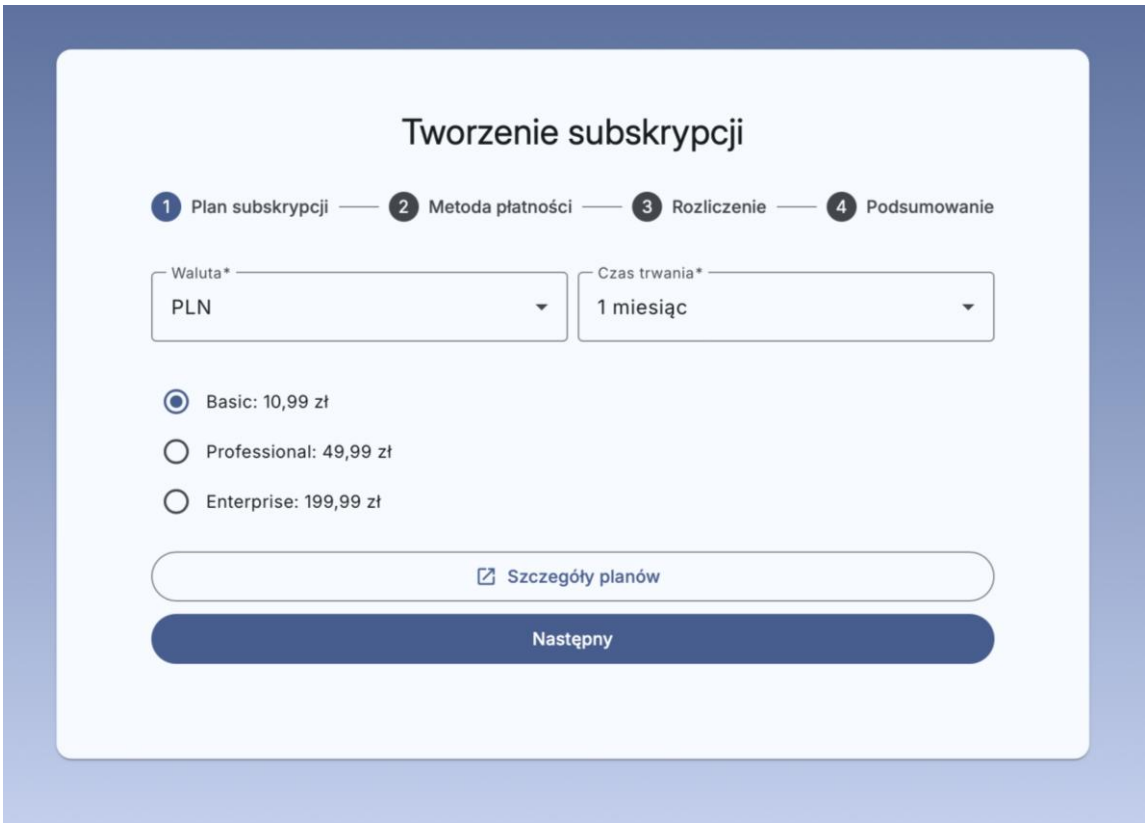


Rysunek 136 Widok odnowienia subskrypcji. Źródło: opracowanie własne.

Kreator subskrypcji

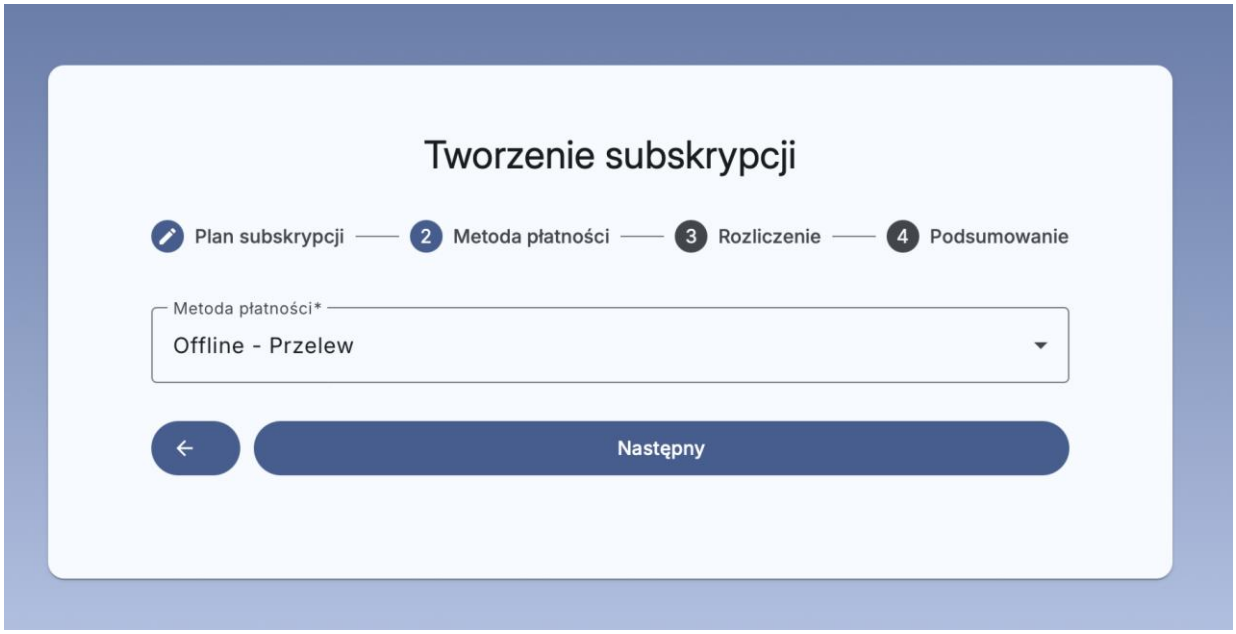
Kreator subskrypcji zrealizowano jako formularz czterokrokowy z wykorzystaniem komponentu *stepper* z Angular Material, wzorując się na innych kreatorach w aplikacji.

Pierwszy krok, przedstawiony na *Rysunek 137*, obejmuje wybór planu. Wydzielono go jako osobny komponent współdzielony z pierwszym krokiem kreatora wypożyczalni, opisanym w podpunkcie *7.2.4.1*.



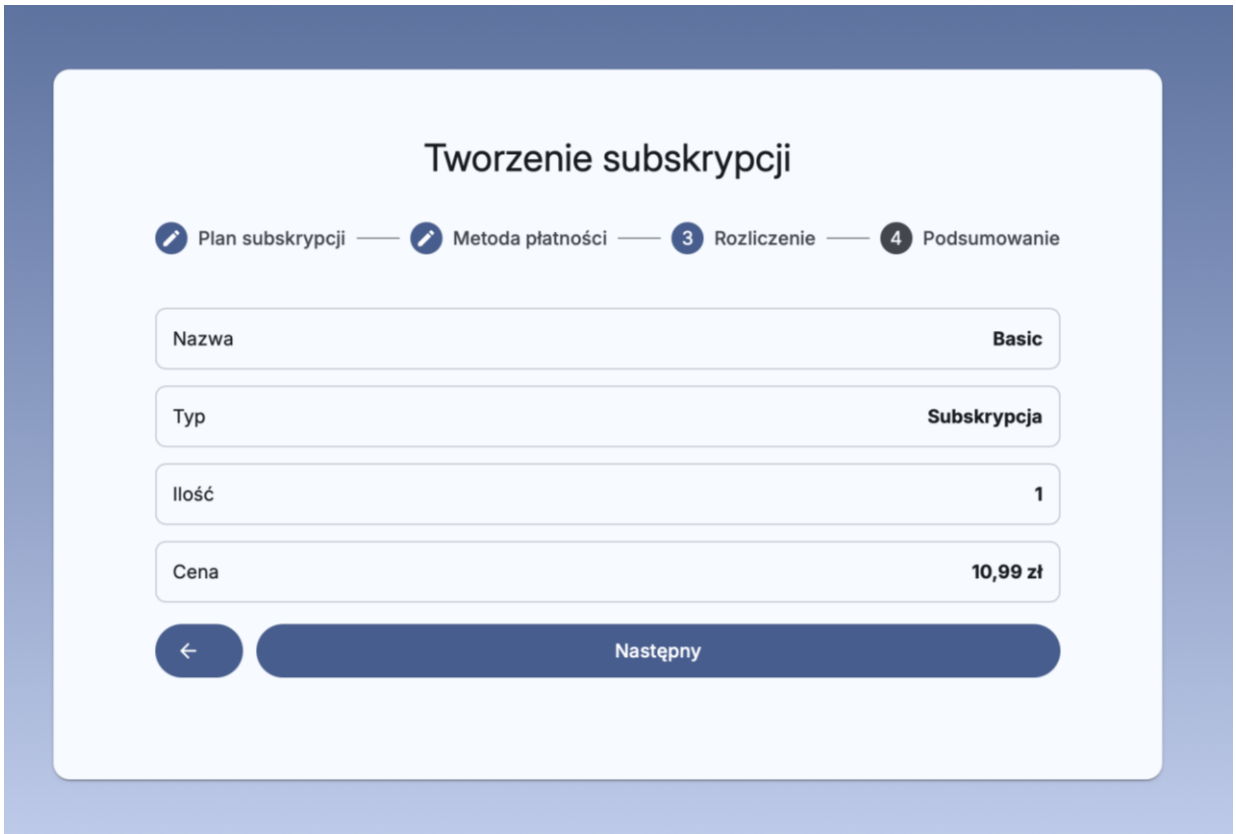
Rysunek 137 Widok kreatora subskrypcji - krok 1. Źródło: opracowanie własne.

Drugi krok, przedstawiony na *Rysunek 138*, obejmuje wybór metody płatności, analogicznie do drugiego kroku kreatora wypożyczalni opisanego w podpunkcie *7.2.4.1*. Krok ten pojawia się jedynie dla planów płatnych.



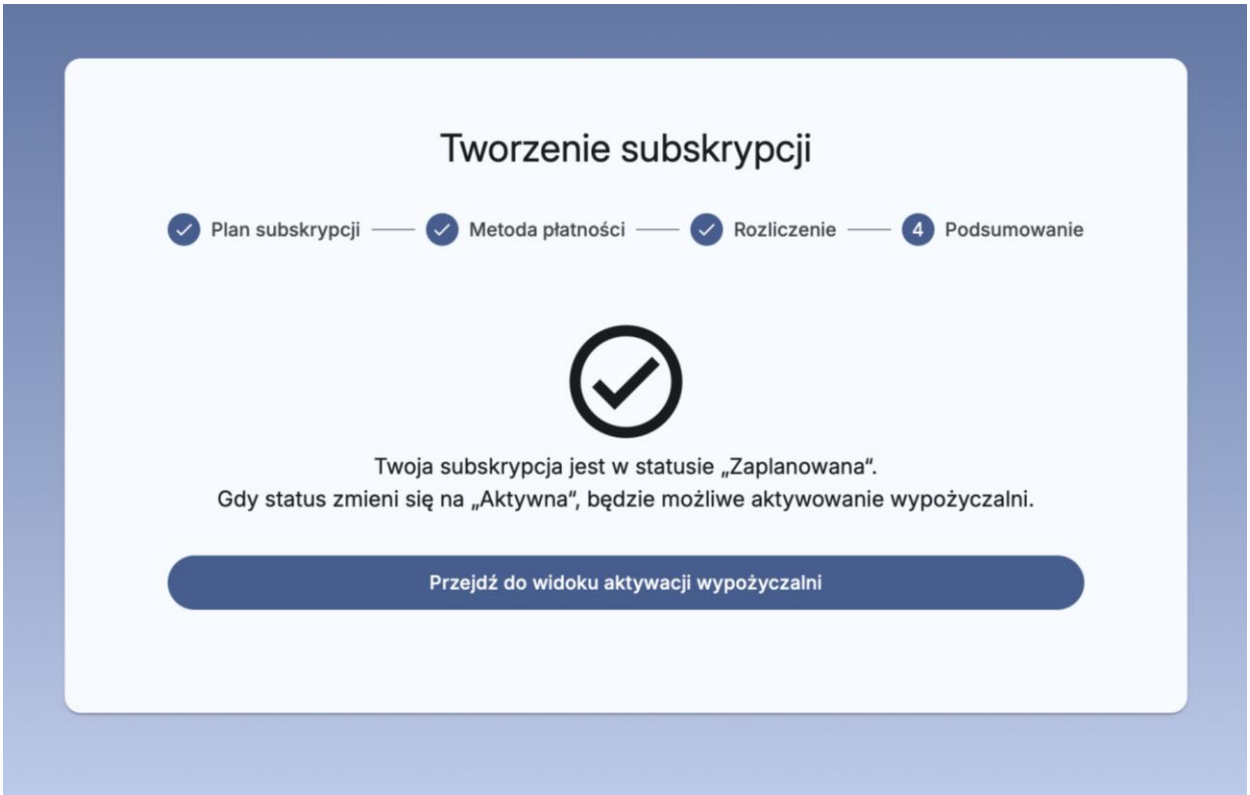
Rysunek 138 Widok kreatora subskrypcji - krok 2. Źródło: opracowanie własne.

Trzeci krok, przedstawiony na *Rysunek 139*, prezentuje podgląd rozliczenia pobierany z serwera przed utworzeniem subskrypcji.



Rysunek 139 Widok kreatora subskrypcji - krok 3. Źródło: opracowanie własne.

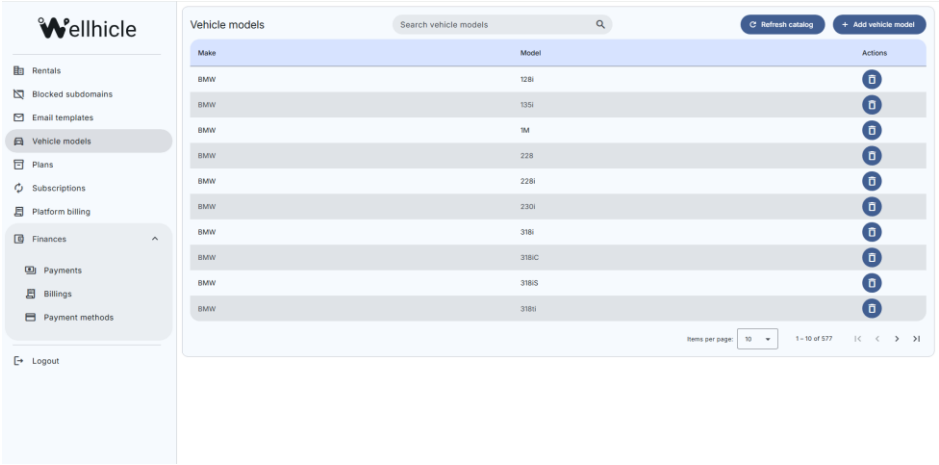
Czwarty krok, przedstawiony na *Rysunek 140*, stanowi ekran podsumowujący.



Rysunek 140 Widok kreatora subskrypcji - krok 4. Źródło: opracowanie własne.

7.2.5. Aplikacja do administrowania wypożyczalniami

Aplikację do administrowania wypożyczalniami przygotowano na bazie aplikacji do zarządzania wypożyczalnią, opisanej w punkcie *7.2.4*. Przykładowy widok aplikacji przedstawiono na *Rysunek 141*.



Rysunek 141 Przykładowy widok aplikacji do administrowania wypożyczalniami. Źródło: opracowanie własne.

Logowanie administratora do wypożyczalni jako właściciel

Rozwiązanie wymaga komunikacji pomiędzy dwiema osobnymi aplikacjami *frontendowymi*: aplikacją do administrowania wypożyczalniami oraz aplikacją do zarządzania wypożyczalnią. Do przekazania danych uwierzytelniających wykorzystano mechanizm `postMessage`.

Po stronie aplikacji administratora najpierw wysyłane jest żądanie do serwera, którego wynikiem jest otrzymanie danych uwierzytelniających dla wskazanej wypożyczalni. Następnie aplikacja otwiera specjalny widok aplikacji wypożyczalni dostępny pod ścieżką */login-admin-as-owner*. Dane uwierzytelniające nie są przekazywane bezpośrednio w adresie URL. Aplikacja administratora oczekuje najpierw na komunikat „*ready*", który informuje, że aplikacja wypożyczalni została uruchomiona i jest gotowa do odebrania danych. Dopiero po otrzymaniu tego komunikatu następuje przekazanie obiektu z danymi uwierzytelniającymi do nowo otwartego okna. Zastosowano również sprawdzenie źródła wiadomości. Aplikacja administratora reaguje wyłącznie na komunikat pochodzący z adresu aplikacji wypożyczalni określonego w konfiguracji środowiska. Po przekazaniu danych nasłuchiwanie na wiadomości zostaje usunięte, ponieważ dalsza komunikacja pomiędzy tymi oknami nie jest już potrzebna. Metoda obsługująca opisywaną funkcję po stronie aplikacji do administrowania wypożyczalniami przedstawiona jest na *Listing 51*.

Listing 51 Metoda otwierająca aplikację wypożyczalni i przekazująca dane uwierzytelniające administratora. Źródło: opracowanie własne.

```
async loginAsOwner(): Promise<void> {
  try {
    const authData = await
firstValueFrom(this.authService.loginAsOwner(this.tenant));
    const newWindow = window.open(`${environment.tenantUrl}/login-admin-
as-owner`, "_blank");

    const handler = (event: MessageEvent): void => {
      if (event.origin === environment.tenantUrl && event.data ===
"ready") {
        newWindow?.postMessage(authData, environment.tenantUrl);
        window.removeEventListener("message", handler);
      }
    };

    window.addEventListener("message", handler);
  } catch (error) {
    if (error instanceof HttpErrorResponse) {
      if (isCustomError(error.error)) {
        this.snackBar.open(error.error.code.toString(), "Close");
      }
    }
  }
}
```

Po stronie aplikacji wypożyczalni przygotowano osobny komponent, którego fragment przedstawiono w *Listing 52*. Po uruchomieniu komponent wysyła do aplikacji administratora komunikat „*ready*". Informuje w ten sposób, że okno aplikacji wypożyczalni zostało załadowane i może odebrać dane. Następnie komponent nasłuchuje na wiadomość zwrotną. Jeżeli wiadomość pochodzi z adresu aplikacji administratora, dane są zapisywane w pamięci operacyjnej serwisu autoryzacji. Po zapisaniu danych użytkownik otrzymuje komunikat potwierdzający, a aplikacja przekierowuje go do widoku rezerwacji.

Listing 52 Fragment komponentu odbierającego dane uwierzytelniające w aplikacji do zarządzania wypożyczalnią. Źródło: opracowanie własne.

```
ngOnInit(): void {
  window.opener.postMessage("ready", environment.adminUrl);
}
```

```
@HostListener("window:message", ["$event"])
async onMessage(event: MessageEvent): Promise<void> {
  if (event.origin !== environment.adminUrl) return;

  this.authService.setInMemoryStorage(event.data as AuthData);
  this.snackBar.open(
    this.translateService.instant("loginAdminAsOwnerComponent.message"),
    this.translateService.instant("close")
  );

  await this.router.navigate(["/", paths.reservations]);
}
```

7.2.6. Aplikacja do korzystania z usług wypożyczalni

Aplikacja do korzystania z usług wypożyczalni stanowi publiczną część systemu przeznaczoną dla klientów wypożyczalni. Opis jej implementacji podzielono na mechanizm rozpoznawania najemcy na podstawie subdomeny, dynamiczne stosowanie motywu kolorystycznego i kroju pisma, proces rezerwacji pojazdu oraz panel klienta z edycją danych i historią rezerwacji.

W odróżnieniu od pozostałych aplikacji przeglądarkowych ten sam kod obsługuje wiele wypożyczalni. Mechanizmy rozpoznawania najemcy i personalizacji wyglądu są zatem niezbędne do prawidłowego wyświetlania danych, oferty i interfejsu przypisanego do konkretnej organizacji.

7.2.6.1. Mechanizm rozpoznawania najemcy

Mechanizm rozpoznawania najemcy rozwiązuje problem uruchamiania jednej aplikacji dla wielu wypożyczalni. Użytkownik nie wybiera wypożyczalni ręcznie - aplikacja ustala go na podstawie adresu strony. Dzięki temu klient wchodzący na właściwą subdomenę otrzymuje dane, ofertę i wygląd przypisane do konkretnej wypożyczalni.

Pierwszym etapem jest odczytanie subdomeny. Funkcja `getTenantDomain` usuwa prefiks `www.`, obsługuje adresy lokalne oraz pozwala wymusić identyfikator najemcy przez `TENANT_OVERRIDE`, co upraszcza uruchamianie aplikacji w środowisku deweloperskim. Fragment przedstawiono na *Listing 53*.

Listing 53 Wyznaczanie subdomeny najemcy. Źródło: opracowanie własne.

```
export function getTenantDomain(): string | null {
    const injectedTenant = (window as { TENANT_OVERRIDE?: string })
        .TENANT_OVERRIDE;
    if (injectedTenant) {
        return injectedTenant;
    }

    const hostname = window.location.hostname.replace(/^www\./, "");
    const parts = hostname.split(".");

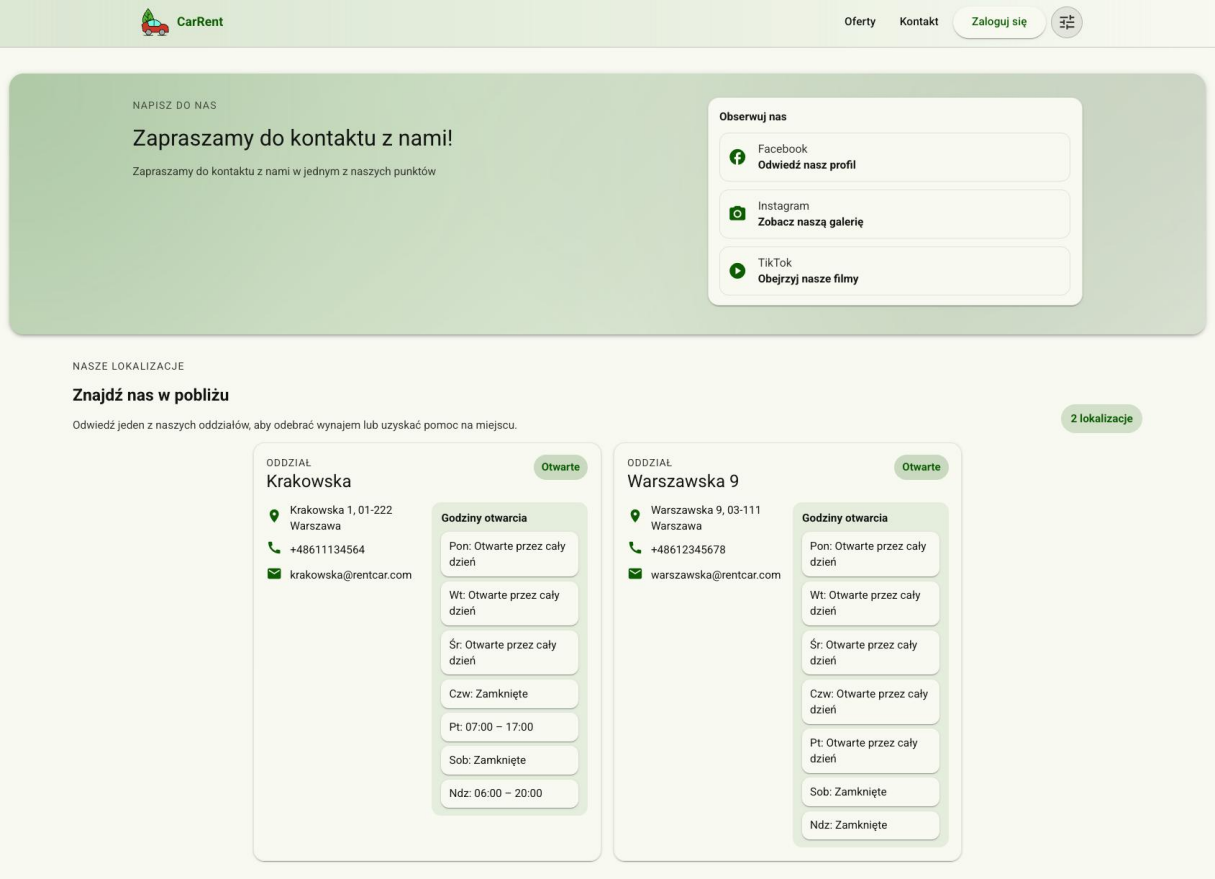
    if (parts.length === 2{
        return parts[0] !== "localhost" ? parts[0] : null;
    }
    return parts.length >= 3 ? parts[0] || null : null;
}
```

Odczytana wartość jest używana przez `OrganizationService`. Serwis przechowuje subdomenę i publiczne dane najemcy w sygnałach, dzięki czemu komponenty mogą odczytywać

aktualny stan bez samodzielnego wykonywania żądań HTTP. Po pobraniu danych ustawiany jest tytuł strony, zapisywana jest kopia danych w pamięci przeglądarki, a następnie aplikowane są ustawienia marki i preferencje organizacji.

Subdomena jest również przekazywana do serwera przy każdym żądaniu HTTP. Odpowiada za to *interceptor*. Nagłówek `X-Tenant-Subdomain` pozwala serwerowi powiązać zapytanie z właściwą organizacją i zwrócić tylko dane należące do tej wypożyczalni.

Na *Rysunek 142* przedstawiono widok strony kontaktowej konkretnego zidentyfikowanego po subdomenie adresu najemcy z załadowanymi danymi o oddziałach.



Rysunek 142 Widok listy ofert uruchomionej pod subdomeną najemcy. Źródło: opracowanie własne.

7.2.6.2. Dynamiczna typografia i kolorystyka aplikacji

Po rozpoznaniu najemcy aplikacja dostosowuje swój wygląd do ustawień wypożyczalni. Zmiana dotyczy głównie motywu kolorystycznego oraz typografii. Mechanizm działa bez przeładowania strony - usługi modyfikują klasy elementu głównego dokumentu, a arkusze stylów reagują na te klasy przez zmienne CSS używane przez Angular Material 3.

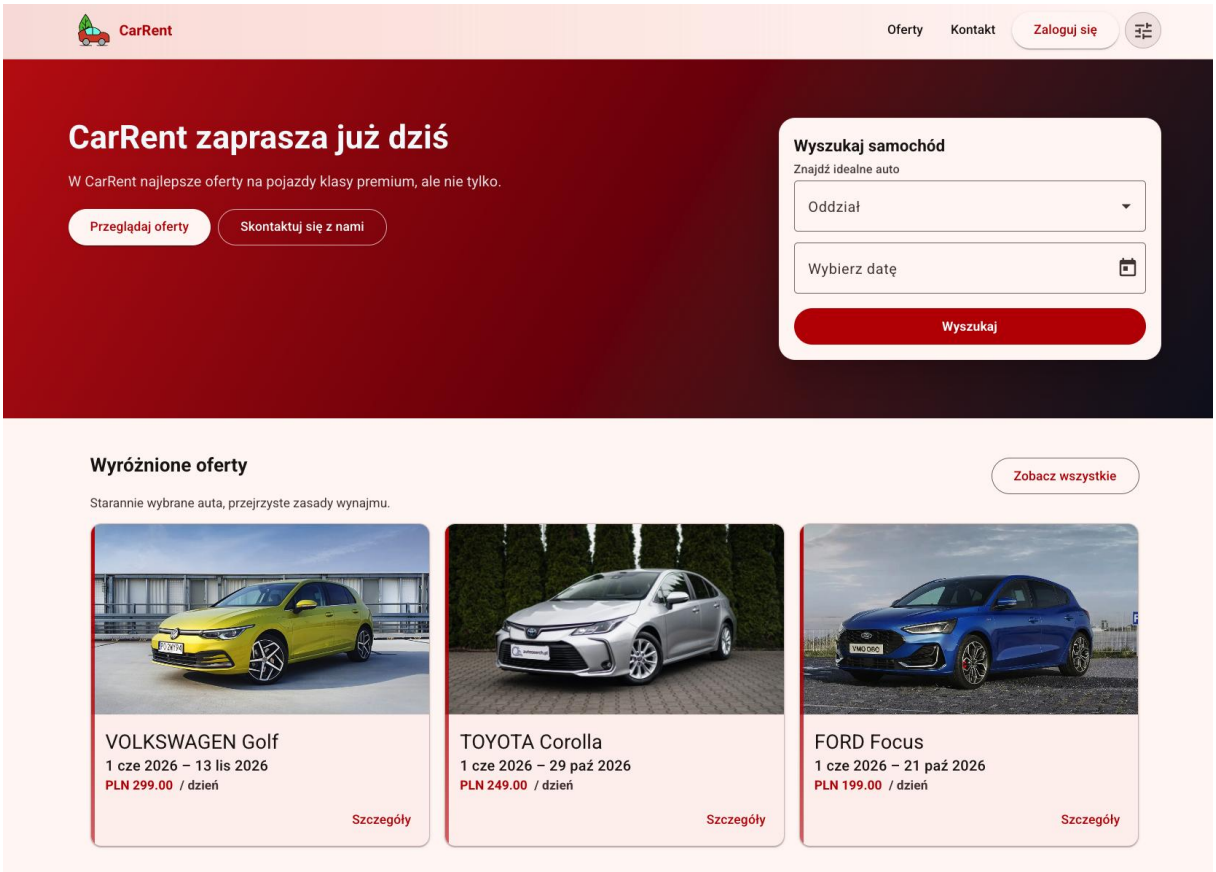
Dane publiczne najemcy zawierają między innymi kolor i krój pisma. `OrganizationService` tłumaczy wartości z odpowiedzi serwera na identyfikatory obsługiwane przez aplikację, a następnie wywołuje usługi odpowiedzialne za motyw i typografię. Aplikacja obsługuje następujące motywy kolorystyczne:

- czerwony,
- zielony,

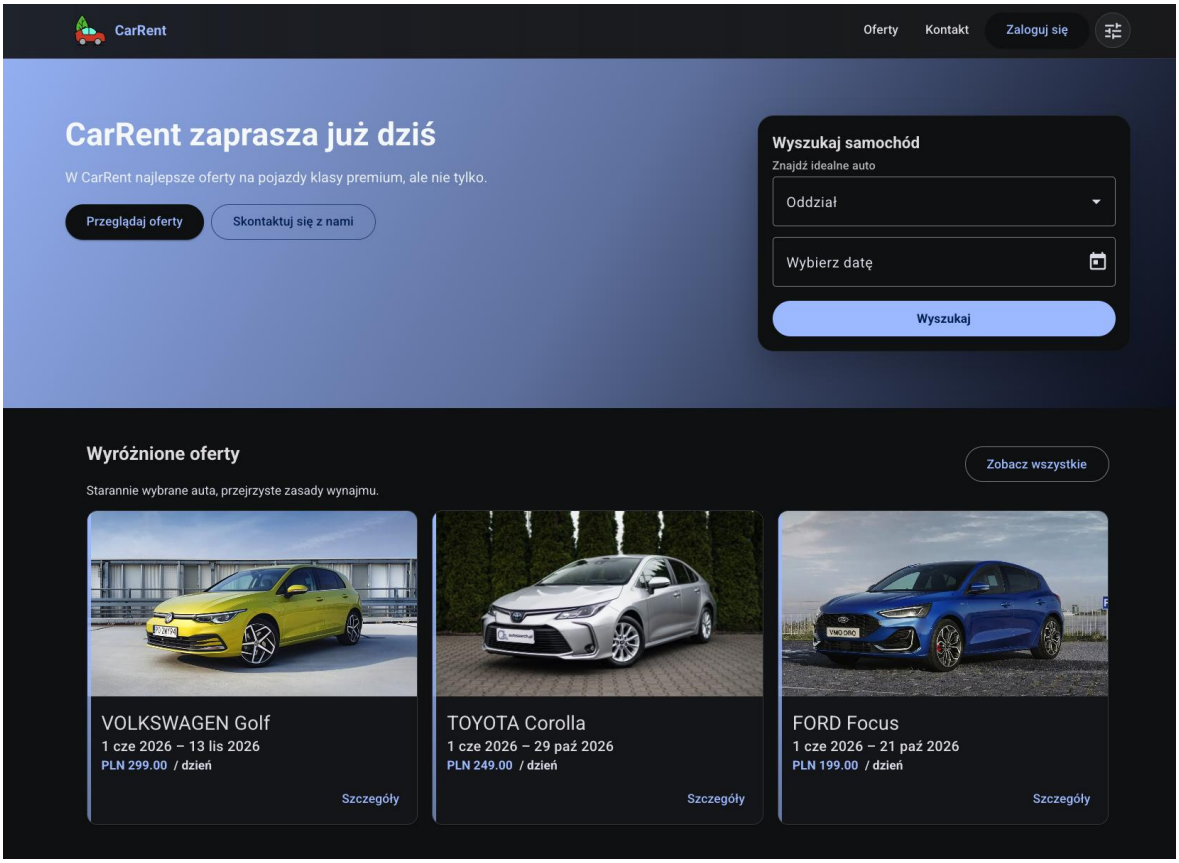
- niebieski,
- żółty,
- cyjan,
- magenta,
- pomarańczowy,
- chartreuse,
- spring-green,
- azure,
- fioletowy,
- różany.

Dodatkowo dostępne są trzy ustawienia trybu: systemowy, jasny i ciemny. Tryb systemowy reaguje na ustawienia przeglądarki użytkownika, dlatego zmiana preferowanego schematu kolorów w systemie może zostać odzwierciedlona w aplikacji na żywo.

Na *Rysunek 143* przedstawiono widok strony głównej aplikacji w kolorystyce czerwonej oraz jasnym motywie. Dla porównania, na *Rysunek 144* przedstawiono tę samą stronę, ale w kolorystyce niebieskiej oraz ciemnym motywie.



Rysunek 143 Widok strony głównej aplikacji w kolorystyce czerwonej w motywie jasnym. Źródło: opracowanie własne.



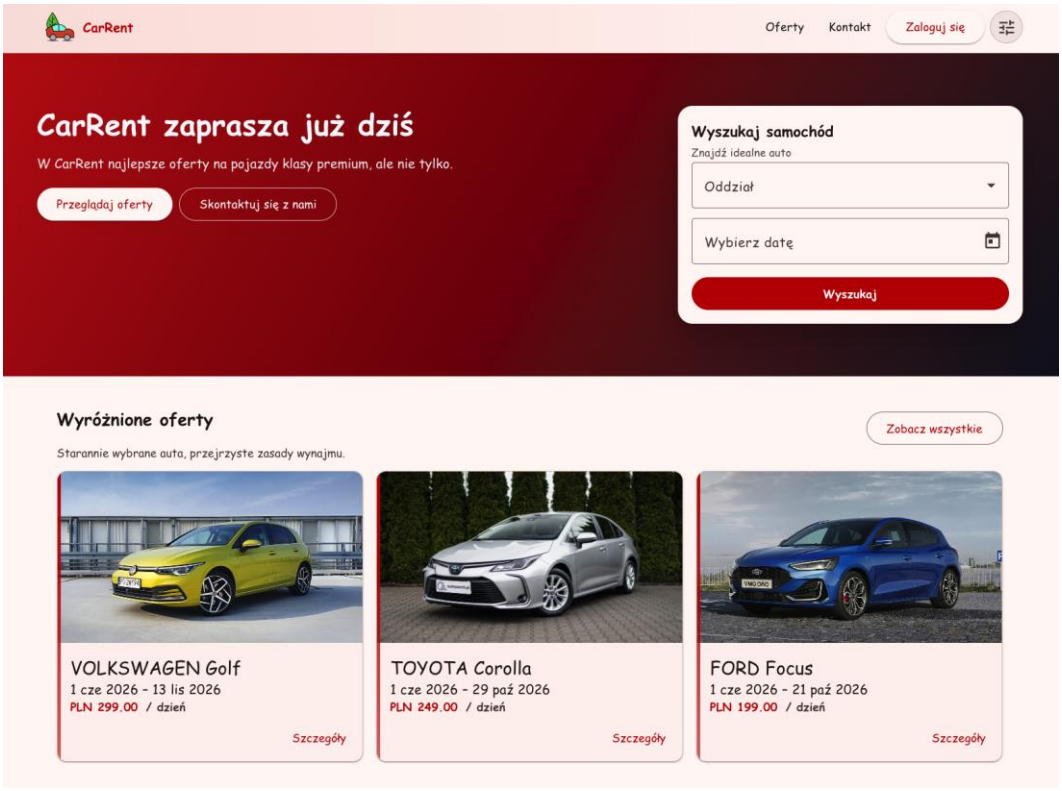
Rysunek 144 Widok strony głównej aplikacji w kolorystyce niebieskiej w motywie ciemnym. Źródło: opracowanie własne.

Analogicznie działa typografia. Dostępne kroje pisma to:

- Inter,
- Roboto,
- Times New Roman,
- Arial,
- Comic Sans.

Zmiana polega na usunięciu poprzedniej klasy zaczynającej się od `font-` i dodaniu nowej klasy odpowiadającej wybranemu krojowi.

Na *Rysunek 145* przedstawiono stronę główną w tej samej kolorystyce co na *Rysunek 143*, ale po zmianie kroju pisma z Roboto na Comic Sans.



Rysunek 145 Widok strony głównej po zmianie kroju pisma na Comic Sans. Źródło: opracowanie własne.

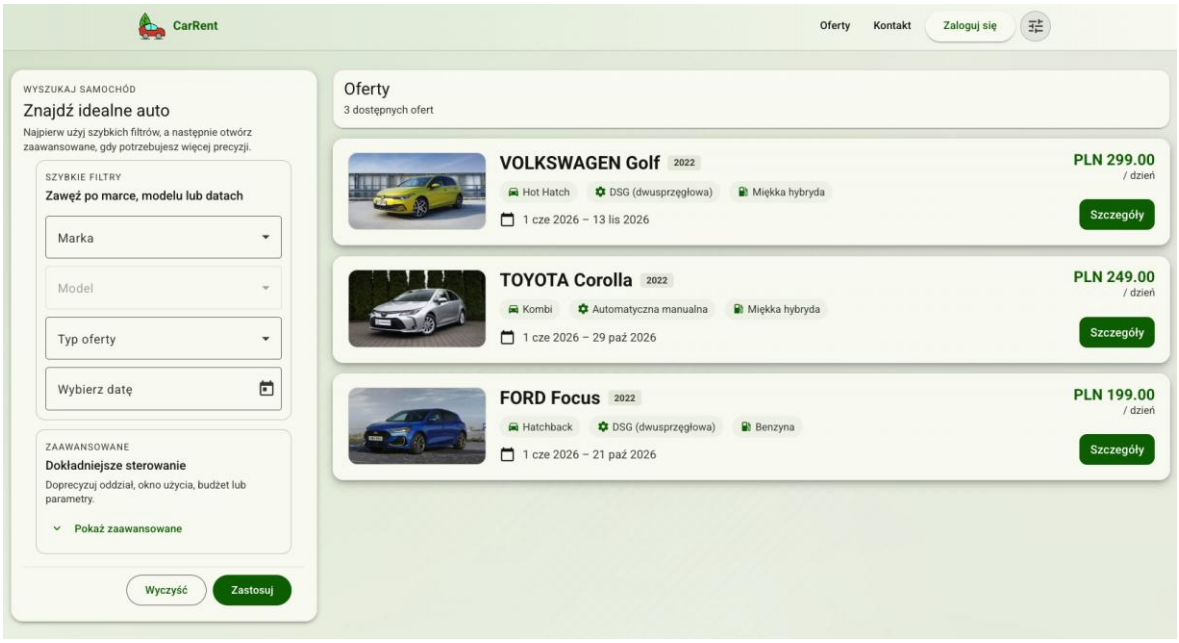
7.2.6.3. Proces rezerwacji pojazdu

Proces rezerwacji prowadzi klienta od listy ofert do utworzenia rezerwacji. Został podzielony na kilka ekranów, aby użytkownik najpierw mógł porównać dostępne pojazdy, następnie sprawdzić szczegóły wybranej oferty, uzupełnić dane rezerwacji i dopiero na końcu wybrać sposób płatności.

Główne kroki procesu są następujące:

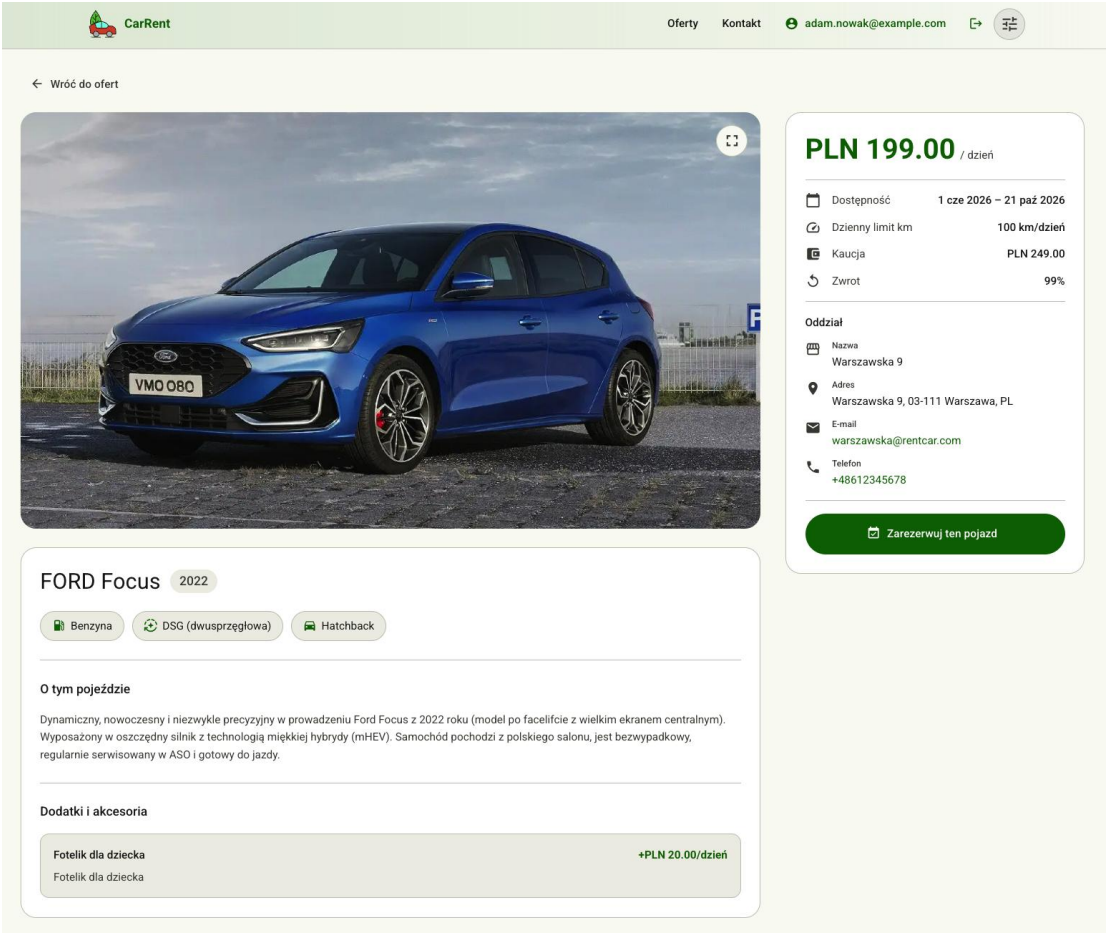
- klient wybiera ofertę z listy,
- aplikacja wyświetla szczegóły pojazdu lub grupy pojazdów,
- użytkownik rozpoczyna rezerwację,
- formularz rezerwacji zbiera przedział czasowy, w którym odbędzie się najem, dane osobowe, adres, dokument osobisty oraz dodatki,
- wersja robocza rezerwacji jest zapisywana w pamięci przeglądarki,
- ekran podsumowania pozwala wybrać sposób płatności,
- po zatwierdzeniu system tworzy rezerwację i przekierowuje klienta do panelu.

Na *Rysunek 146* przedstawiono widok listy ofert pojazdów dla przykładowego najemcy CarRent.



Rysunek 146 Widok listy ofert z filtrami. Źródło: opracowanie własne.

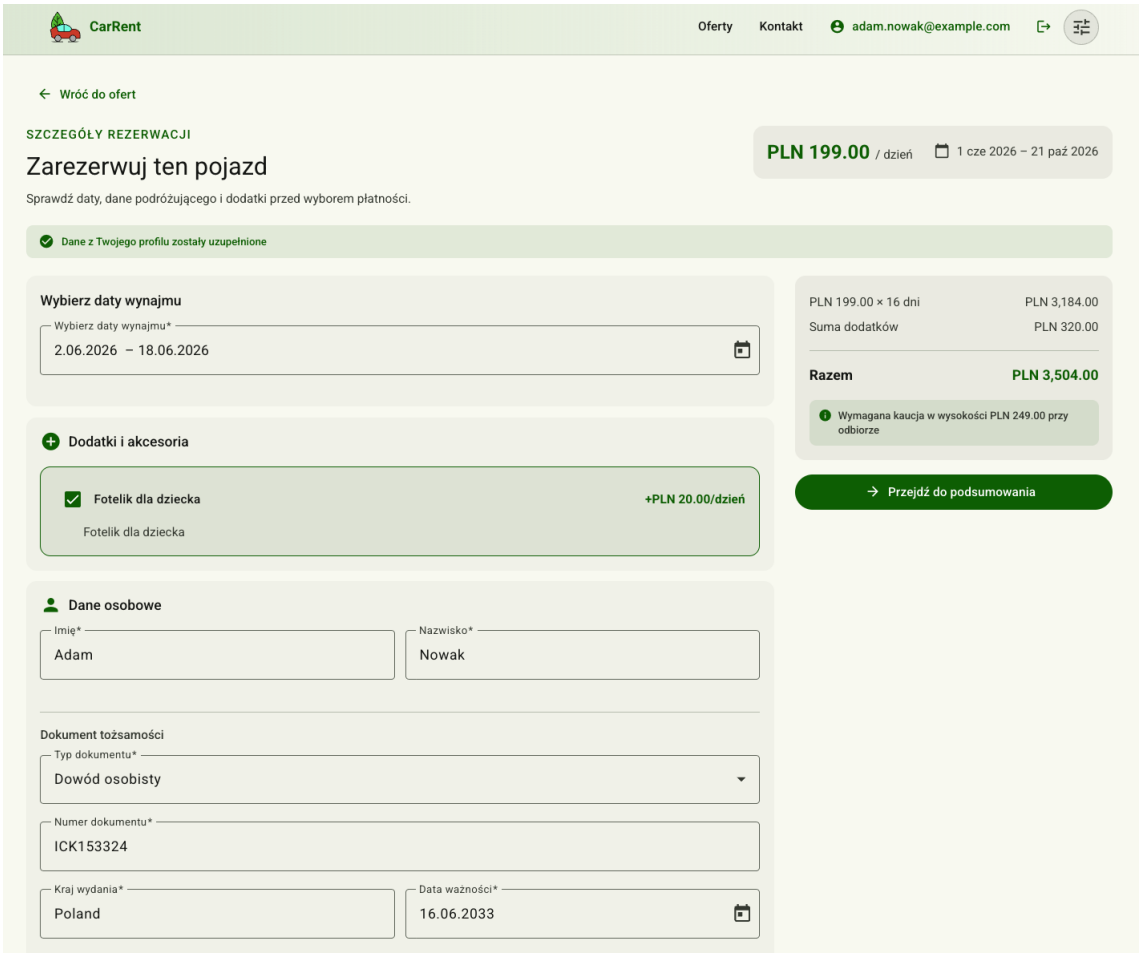
Rozpoczęcie rezerwacji następuje w widoku szczegółów oferty. Jeżeli klient nie jest zalogowany, aplikacja kieruje go na ekran logowania, a po zalogowaniu przekierowuje na docelową stronę. Dzięki temu po uwierzytelnieniu użytkownik może kontynuować proces od tej samej oferty. Widok szczegółów oferty pojazdu został przedstawiony na *Rysunek 147*.



Rysunek 147 Widok szczegółów oferty z przyciskiem rozpoczęcia rezerwacji. Źródło: opracowanie własne.

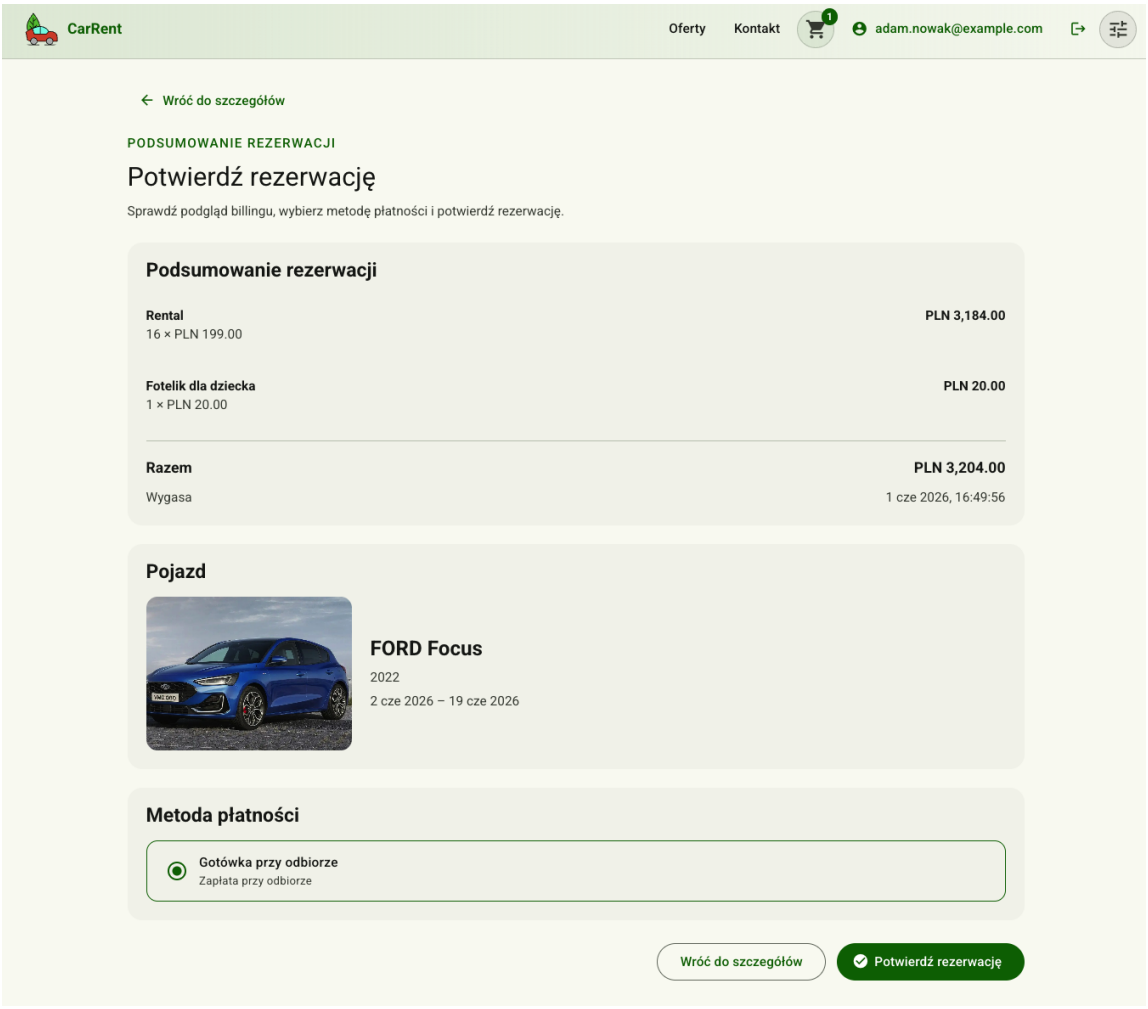
Formularz rezerwacji sprawdza poprawność danych, buduje żądanie utworzenia rezerwacji i zapisuje je jako szkic. Szkic jest potrzebny, ponieważ wybór płatności odbywa się na kolejnym ekranie. Po zapisaniu danych aplikacja przechodzi do podsumowania. Za utrwalenie szkicu odpowiada `ReservationFlowService`. Dane są zapisywane pod kluczem zawierającym identyfikator oferty, co pozwala rozróżnić kilka rozpoczętych rezerwacji.

Widok fragmentu formularza rezerwacyjnego z wybranym dodatkiem, datą najmu oraz wypełnioną sekcją z dokumentem tożsamości przedstawiono na *Rysunek 148*.



Rysunek 148 Fragment formularza rezerwacji. Źródło: opracowanie własne.

Na ekranie podsumowania klient wybiera metodę płatności. Wybrana metoda zostaje dopisana do szkicu, a następnie aplikacja pobiera podgląd rozliczenia. Po zatwierdzeniu rezerwacji wersja robocza jest usuwana, a użytkownik trafia do panelu klienta. Widok podsumowania rezerwacji z wyborem metody płatności został przedstawiony na *Rysunek 149*

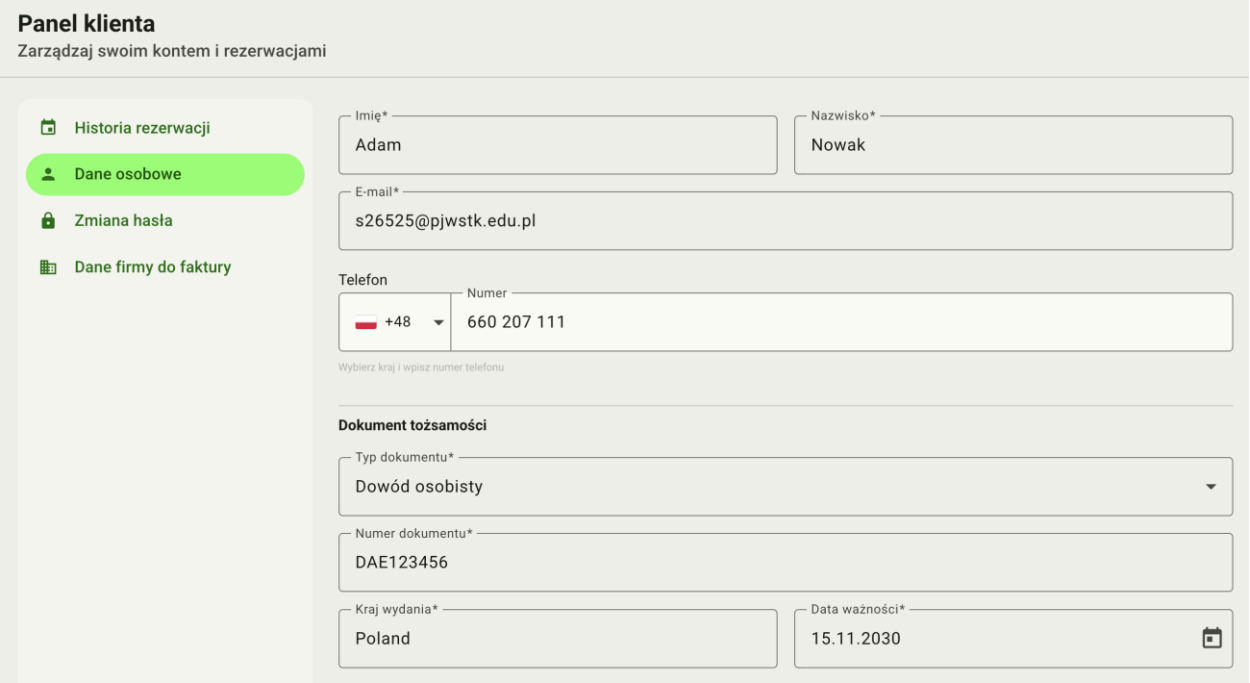


Rysunek 149 Widok podsumowania rezerwacji. Źródło: opracowanie własne.

7.2.6.4. Panel klienta

Panel klienta grupuje funkcje dostępne po zalogowaniu. Użytkownik może w nim zmienić dane osobowe, numer telefonu, adres, dane dokumentu oraz dane firmowe. Druga część panelu prezentuje historię rezerwacji i pozwala sprawdzić szczegóły wybranej pozycji.

Po wejściu do panelu aplikacja pobiera bieżące dane klienta z serwera i uzupełnia formularze. Ten sam formularz jest używany zarówno do wyświetlenia aktualnych danych, jak i do ich późniejszej edycji. Na *Rysunek 150* przedstawiono fragment widoku formularza edycji danych osobowych w panelu klienta z przykładowymi danymi.

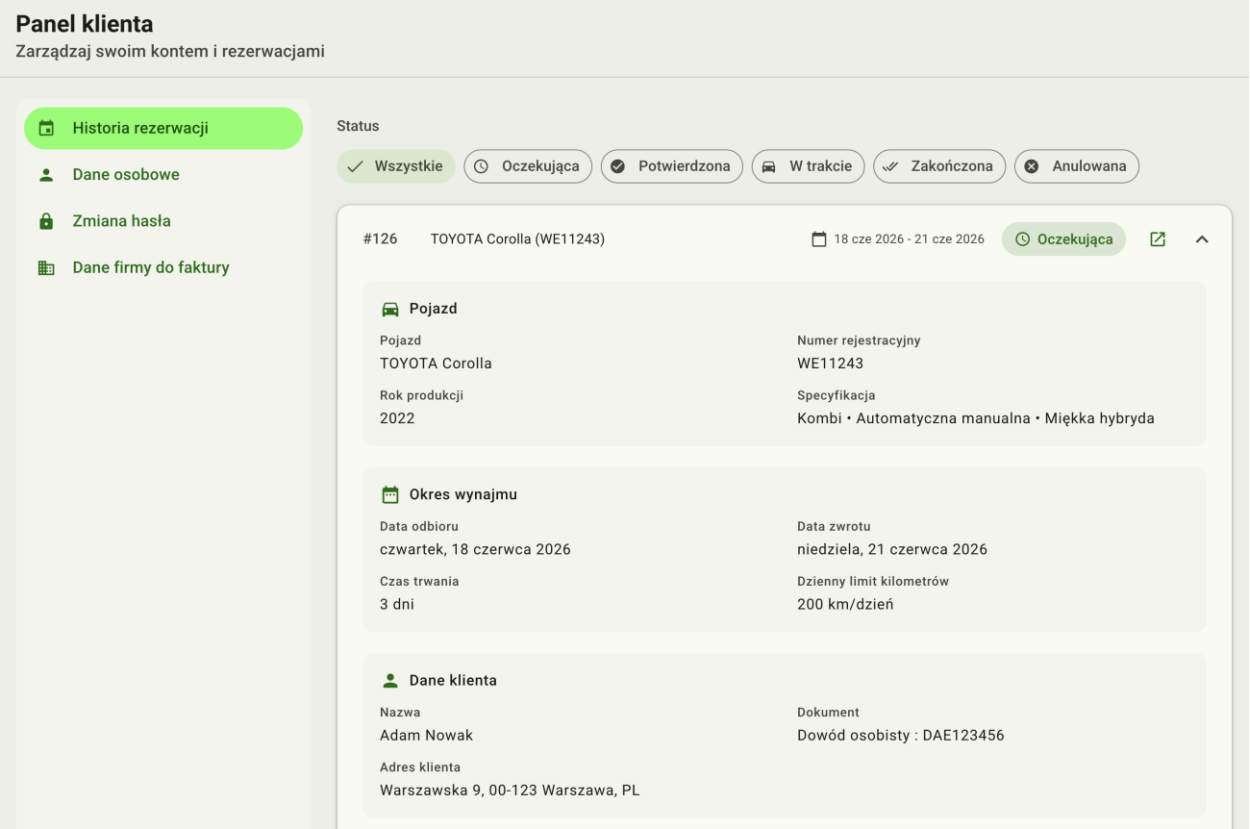


Rysunek 150 Widok formularza danych osobowych w panelu klienta. Źródło: opracowanie własne.

Zapis danych profilu polega na zbudowaniu obiektu zawierającego dane osobowe, numer telefonu, adres oraz dokument tożsamości. Puste pola opcjonalne są zamieniane na wartości undefined albo null, aby serwer otrzymał jednoznaczną informację, które dane należy zachować, a które wyczyścić.

Historia rezerwacji jest obsługiwana przez ReservationsService. Serwis pobiera listę rezerwacji z uwzględnieniem stronicowania, przechowuje stan ładowania i utrzymuje pamięć podręczną szczegółów. Szczegóły pojedynczej rezerwacji są pobierane dopiero po rozwinięciu elementu w panelu. Dzięki temu pierwszy widok historii jest lżejszy, a dodatkowe dane są pobierane tylko wtedy, gdy klient rzeczywiście chce je zobaczyć.

Na *Rysunek 151* przedstawiono fragment widoku szczegółów rezerwacji w panelu klienta.



Rysunek 151 Widok rozwiniętych szczegółów rezerwacji. Źródło: opracowanie własne.

7.3. Interfejs mobilny

Aplikacja mobilna Wellhicle stanowi dedykowane narzędzie dla pracowników operacyjnych wypożyczalni realizujących procesy terenowe: wydanie pojazdu klientowi, przyjęcie zwracanego pojazdu po zakończeniu najmu oraz zgłaszanie usterek wykrytych podczas oględzin. Aplikacja działa na platformach Android i iOS, wymieniając dane z serwerem aplikacyjnym Wellhicle przez interfejs REST API. Podrozdział opisuje decyzje technologiczne, architekturę modułową, implementację kluczowych procesów biznesowych oraz rozwiązania wynikające z wieloplatformowego charakteru projektu.

7.3.1. Nawigacja

Nawigacja w aplikacji opiera się na bibliotece navigation-compose v2.9.1 i jest zorganizowana wokół komponentu *RootNavHost*. Przełączanie między częścią autoryzacyjną (`AuthRoot`) a częścią właściwą (`MainRoot`) odbywa się na podstawie stanu sesji użytkownika – obecności ważnego tokenu JWT w `AuthStorage`, obserwowanego przez `LoginViewModel`, jak przedstawiono na *Listing 54*.

Listing 54 Nawigacja najwyższego poziomu reagująca na zmianę stanu sesji – wylogowanie czyści cały stos nawigacyjny. Źródło: opracowanie własne.

```
// commonMain - RootNavHost.kt
@Composable
fun RootNavHost(
    loginViewModel: LoginViewModel,
    navController: NavHostController,
    topBarStateHolder: MutableState<TopBarState>
) {
```

```
    val isAuthenticated = loginViewModel.uiState.isAuthenticated

    LaunchedEffect(isAuthenticated) {
        if (!isAuthenticated) {
            navController.navigate(AuthRoot) {
                popUpTo(0) { inclusive = true }
                launchSingleTop = true
            }
        }
    }

    NavHost(
        navController = navController,
        startDestination = if (isAuthenticated) MainRoot else AuthRoot
    ) {
        navigation(startDestination = "login.main", route = AuthRoot) {
            authNavGraph(loginViewModel, navController, onAuthenticated =
{
                navController.navigate(MainRoot) {
                    popUpTo(AuthRoot) { inclusive = true }
                    launchSingleTop = true
                }
            })
        }
        navigation(startDestination = Screen.CarHandover.route, route =
MainRoot) {
            mainNavGraph(navController, topBarStateHolder)
        }
    }
}
```

Podejście to centralizuje decyzję o routingu w jednym miejscu, eliminując konieczność sprawdzania stanu logowania w każdym ekranie. Wylogowanie – niezależnie od tego, który ekran jest aktywny – powoduje nawigację do *AuthRoot* z usunięciem całego stosu nawigacyjnego (*popUpTo(0)*).

Trasy nawigacyjne zostały zdefiniowane jako *sealed class Screen* w module *core:ui*, dzięki czemu wszystkie moduły *feature* korzystają z jednego zestawu tras. Podejście to zapobiega duplikacji literałów tekstowych i ułatwia refaktoryzację tras bez ryzyka niespójności między modułami.

Część właściwa aplikacji (*MainRoot*) udostępnia cztery zakładki dolnego paska nawigacyjnego: Zgłoszenia usterek (*Services*), Wydanie pojazdu (*CarHandover*), Zwrot pojazdu (*CarReturn*) oraz Profil pracownika – dostępny przez górny pasek nawigacyjny. Dla procesów wieloekranowych (*handover*, *carreturn*) zastosowano zagnieżdżone grafy nawigacyjne (*HandoverNavGraph*, *ReturnNavGraph*). *ViewModel* dla każdego procesu jest tworzony z zasięgiem całego grafu nawigacyjnego, a nie pojedynczego ekranu – dzięki temu dane zebrane na wcześniejszych krokach (rezerwacja, zdjęcia, wyniki kontroli) są dostępne na każdym etapie procesu bez konieczności ich przekazywania przez parametry tras.

Przejścia między ekranami procesów wielokrokowych realizowane są z animacją *fadeIn*/*fadeOut*. Efekty nawigacyjne (zdarzenia jednorazowe) są zbierane w ramach pojedynczego *LaunchedEffect(Unit)* per ekran w grafie (*CollectEffectsOnce*), co eliminuje ryzyko powtórnej obsługi tego samego zdarzenia przy rekompozycji.

7.3.2. Struktura aplikacji

Aplikacja mobilna podzielona jest na moduły *Gradle* zarządzane plikiem *settings.gradle.kts*:

- `composeApp` – punkt wejścia, nawigacja, inicjalizacja *Koin*
- `core:api` – klient HTTP (*Ktor*), modele API, *TokenProvider*
- `core:ui` – wspólne komponenty *Compose*, mechanizmy platformowe
- `core:files` – operacje na plikach (*okio*)
- `shared` – repozytoria, AuthStorage, UserSettingsStorage
- `feature:auth` – logowanie, przypomnienie hasła
- `feature:handover` – wydanie pojazdu (wizard wieloekranowy)
- `feature:carreturn` – zwrot pojazdu
- `feature:services` – zgłaszanie usterek
- `feature:profile` – profil pracownika

Podział na moduły wynika z zasady separacji odpowiedzialności i pozwala na niezależną kompilację oraz testowanie każdej części systemu.

Moduł composeApp stanowi punkt wejścia aplikacji. Zawiera `MainActivity` (Android) i `MainViewController` (iOS), inicjalizuje kontenery Koin oraz opakowanie lokalizacyjne `WithAppLocale`. Nawigacja najwyższego poziomu (`RootNavHost`) jest zdefiniowana w tym module.

Moduł core:api zawiera klienta HTTP (`ApiClientService` – *Ktor*), obsługę tokenów (`TokenProvider`, `StorageBackedTokenProvider`), modele odpowiedzi API oraz konfigurację środowiska (`AppEnvironment`). Wydzielenie warstwy sieciowej do osobnego modułu umożliwia jej wymianę lub podstawienie atrap testowych bez wpływu na pozostałe moduły.

Moduł core:ui grupuje komponenty Compose wielokrotnego użytku: układ kroków procesu (`StepLayout`), kartę rezerwacji (`ReservationCard`), obsługę zdjęć (`PhotosComponent`), pole podpisu cyfrowego (`SignatureComponent`), a także implementacje specyficzne platformowo (`PhotoProvider, LockLandscapeEffect, PdfViewer`).

Moduł core:files dostarcza abstrakcji operacji plikowych opartej na `FileSystem` z biblioteki *okio 3.16.4*, używanej do zapisu danych gdy wysłanie do API zakończy się niepowodzeniem.

Moduł shared zawiera repozytoria dostępu do danych (`AuthRepository`, `UserProfileRepository`, `VehicleModelRepository`) oraz warstwy przechowywania danych: `AuthStorage` (tokeny JWT) i `UserSettingsStorage` (preferencje, np. język) – oparte o `multiplatform-settings`.

Moduły feature:* implementują poszczególne ekrany i procesy biznesowe. Każdy moduł zawiera: ekrany `Compose` (warstwa widoku), `ViewModel` (warstwa logiki prezentacji), mappery (izolacja modeli API od modeli UI) oraz moduł *Koin* z definicją zależności.

7.3.3. Konwencjonalne wtyczki Gradle

Konfiguracja modułów *Gradle* jest standaryzowana przez dwie wtyczki konwencji zdefiniowane w podprojekcie *convention*:

ModulePlugin – bazowa konfiguracja KMP: cele kompilacji Android/iOS (*x64, arm64, simulatorArm64),* ustawienia JVM 17, domyślna zależność *kotlinx-coroutines*. Każdy nowy moduł biblioteczny stosuje tę wtyczkę w swoim *build.gradle.kts* przez *alias(libs.plugins.module.plugin)*. Implementację wtyczki przedstawiono na *Listing 55.*

Listing 55 Bazowa wtyczka konwencji Gradle konfigurująca cele kompilacji KMP dla Androida i iOS. Źródło: opracowanie własne.

```
// convention/ModulePlugin.kt (fragment)
class ModulePlugin : Plugin<Project> {

    override fun apply(target: Project) = with(target) {
        val libs = the<LibrariesForLibs>()
        pluginManager
            .apply(libs.plugins.kotlinMultiplatform.get().pluginId)
        pluginManager
            .apply(libs.plugins.androidLibrary.get().pluginId)

        extensions.getByType<KotlinMultiplatformExtension>().apply {
                androidTarget {
                    compilerOptions { jvmTarget.set(JvmTarget.JVM_17) }
                    }
                listOf(iosX64(), iosArm64(), iosSimulatorArm64())
                sourceSets.getByName("commonMain") {
                    dependencies {
                        implementation(libs.kotlinx.coroutines)
                    }
            }
        }
    }
}
```

FeaturePlugin – rozszerza `ModulePlugin` o automatyczne zależności do core:ui, shared, bibliotek *Koin (compose, viewmodel, navigation)* i *lifecycle-viewmodel-compose*. Dzięki temu każdy moduł `feature` nie musi ręcznie deklarować tych samych zależności, jak przedstawiono na Listing 56.

Listing 56 Wtyczka konwencji Gradle dodająca wspólne zależności do wszystkich modułów feature. Źródło: opracowanie własne.

```
// convention/FeaturePlugin.kt (fragment)
class FeaturePlugin : Plugin<Project> {
    override fun apply(target: Project) = with(target) {
        pluginManager.apply("module.plugin")
        val libs = the<LibrariesForLibs>()
        extensions.getByType<KotlinMultiplatformExtension>().sourceSets
            .getByName("commonMain") {
                dependencies {
                    implementation(project(":core:ui"))
                    implementation(project(":shared"))
                    implementation(libs.koin.di.compose
                        .viewmodel.navigation)
                    implementation(libs.koin.di.compose.viewmodel)
                    implementation(libs.koin.di.compose)
                    implementation(libs.androidx.lifecycle
                        .viewmodelCompose)
                    implementation(libs.kotlinx.coroutines)
                }
```

```
            }
        }
    }
```

Zastosowanie wtyczek konwencji eliminuje powielanie identycznej konfiguracji w build.gradle.kts każdego modułu – przy ośmiu modułach feature i core redukuje to objętość plików konfiguracyjnych i minimalizuje ryzyko niespójności wersji zależności między modułami.

7.3.4. Procesy biznesowe

Aplikacja mobilna obsługuje cztery główne procesy biznesowe: logowanie i zarządzanie sesją, wydanie pojazdu (UC-OPE-01), zwrot pojazdu (UC-OPE-02) oraz zgłaszanie usterek (UC-OPE-03) – szczegółowy opis wymienionych wymagań funkcjonalnych znajduje się w podpunkcie 5.3.8.2. Procesy wydania i zwrotu pojazdu zostały zrealizowane jako wielokrokowe przepływy według wzorca wizard, co pozwala na prowadzenie pracownika przez kolejne etapy i walidację danych na bieżąco.

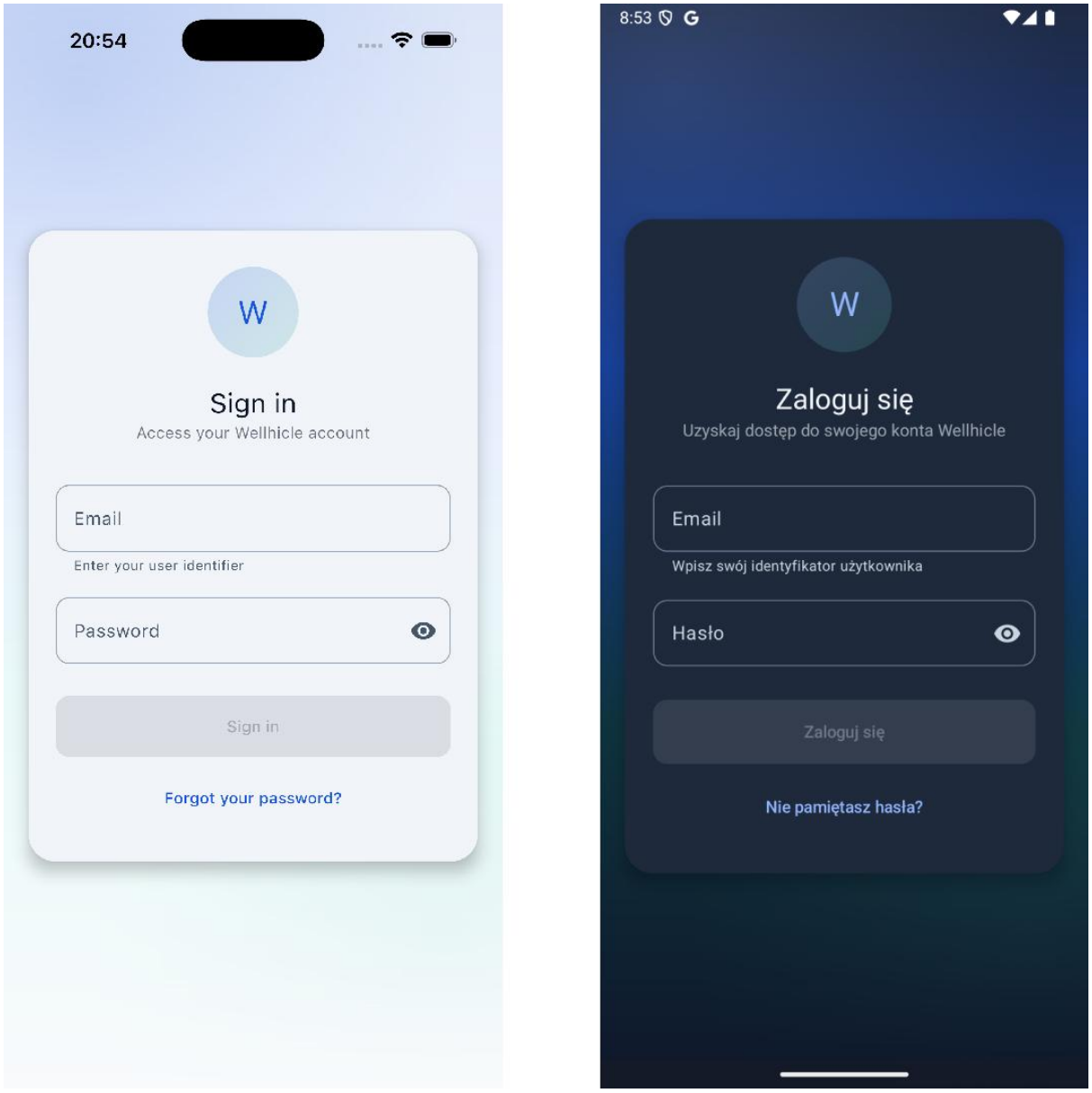
7.3.4.1. Logowanie i zarządzanie sesją

Proces uwierzytelniania obsługiwany jest przez moduł `feature:auth`, który zawiera trzy ekrany: `LoginScreen, ForgotPasswordScreen i ResetPasswordScreen`. Logiką sesji zarządza `LoginViewModel` (w *composeApp*) we współpracy z `AuthRepository` (w module *shared*).

Po uruchomieniu aplikacji `LoginViewModel` inicjalizuje stan sesji i wywołuje `validateSessionOnStart()`, która próbuje odświeżyć token przed wyświetleniem jakiegokolwiek ekranu. Jeśli odświeżenie zakończy się sukcesem, użytkownik zostaje skierowany bezpośrednio do części właściwej aplikacji z pominięciem ekranu logowania, jak przedstawiono na *Listing 57* i *Rysunek 152*.

Listing 57 Odświeżanie tokenu sesji przy każdym starcie aplikacji – sukces pomija ekran logowania. Źródło: opracowanie własne.

```
// AuthRepository.kt
override suspend fun validateSessionOnStart(): Result<LoginResponse> {
    val res = api.refresh()
    return res.onSuccess { lr ->
        storage.saveTokens(
            accessToken = lr.accessToken,
            refreshToken = lr.refreshToken,
            expiresInSeconds = lr.expiresInSec,
            email = lr.email
        )
        _isAuthenticated.value = true
        scheduleNextRefresh()
    }
}
```



Rysunek 152 Ekran logowania aplikacji Wellhicle. Źródło: opracowanie własne.

Po pomyślnym zalogowaniu `AuthRepository` zapisuje tokeny w `AuthStorage` i uruchamia mechanizm automatycznego odświeżania (`startAutoRefresh`). Zadanie odświeżające jest planowane z wyprzedzeniem jednej minuty względem czasu wygaśnięcia tokenu dostępu, jak przedstawiono na *Listing 58*.

Listing 58 Harmonogram automatycznego odświeżania tokenu z minutowym wyprzedzeniem przed jego wygaśnięciem. Źródło: opracowanie własne.

```
// AuthRepository.kt (fragment scheduleNextRefresh)
val delayMillis = max(0L, (expiresAt - oneMinuteMillis) - nowMillis)
delay(max(500L, delayMillis))
val result = refresh()
result.onFailure { _isAuthenticated.value = false }
```

Błędy logowania są klasyfikowane według kodu HTTP odpowiedzi: status 401 mapowany jest na `LoginError.UNAUTHORIZED`, a status 400 na `LoginError.BAD_REQUEST`, co pozwala wyświetlać precyzyjne komunikaty na ekranie logowania. Stan sesji (`isAuthenticated`) jest

eksponowany jako `StateFlow<Boolean>`, co umożliwia reaktywne przełączanie między grafami nawigacyjnymi w `RootNavHost`.

Opcja przypomnienia hasła realizowana jest przez formularz z adresem e-mail (`ForgotPasswordScreen`) i ekran resetowania hasła tokenem (`ResetPasswordScreen`) – oba przepływy wywołują odpowiednie punkty końcowe API przez `AuthRepository.forgotPassword()` i `AuthRepository.resetPassword()`.

7.3.4.2. Wydanie pojazdu (UC-OPE-01)

Proces wydania pojazdu realizuje przypadek użycia UC-OPE-0-01 i obejmuje sześć ekranów prowadzonych przez `CarHandoverViewModel`. Stan całego procesu jest reprezentowany jako jedna niemutowalna klasa `CarHandoverUiState`, jak przedstawiono na *Listing 59*:

Listing 59 Niemutowalna klasa stanu procesu wydania pojazdu obejmująca wszystkie etapy wielokrokowego wizarda. Źródło: opracowanie własne.

```
data class CarHandoverUiState(
    val handovers: List<Reservation> = emptyList(),
    val isLoading: Boolean = false,
    val isRefreshing: Boolean = false,
    val isLoadingMore: Boolean = false,
    val isError: Boolean = false,
    val errorMessage: String? = null,

    val currentPage: Int = 1,
    val hasNextPage: Boolean = false,

    val selectedReservation: Reservation? = null,

    // weryfikacja klienta
    val customerVerified: Boolean = false,
    val customerDocNumber: String = "",
    val customerDocIssuingCountry: String = "",
    val customerVerificationErrors: Map<String, String> = emptyMap(),

    // inspekcja
    val odometerKm: Int? = null,
    val fuelLevel: Int = 3,
    val tireCondition: Int = 10,
    val bodyCondition: Int = 5,
    val photos: List<PhotoItem> = emptyList(),
    val damageNotes: String = "",
    val inspectionErrors: Map<String, String> = emptyMap(),

    // kontrakt / podpis
    val contractHtml: String? = null,
    val signatureSaved: Boolean = false,
    val isSignatureFieldEnabled: Boolean = false,
    val showMissingSignatureDialog: Boolean = false
)
```

Zdarzenia nawigacyjne między ekranami są emitowane jako `SharedFlow<CarHandoverEffect>`, co oddziela logikę nawigacji od stanu UI i umożliwia jednokrotne przetworzenie każdego zdarzenia, jak przedstawiono na *Listing 60*.

Listing 60 Zdarzenia nawigacyjne procesu wydania pojazdu emitowane jako SharedFlow dla jednokrotnego przetworzenia. Źródło: opracowanie własne.

```
sealed interface CarHandoverEffect {
    data class ToVerify(val reservationId: Long)      : CarHandoverEffect
```

```
    data class ToInspection(val reservationId: Long) : CarHandoverEffect
    data class ToContract(val reservationId: Long)   : CarHandoverEffect
    data class ToSignature(val reservationId: Long)  : CarHandoverEffect
    data object ToCompleted : CarHandoverEffect
    data object BackToList  : CarHandoverEffect
    data object ToProfile   : CarHandoverEffect
}
```

Krok 1 – lista rezerwacji. `CarHandoverViewModel` pobiera przy inicjalizacji listę aktywnych rezerwacji ze statusem oczekującym na wydanie przez `ApiClientService.getAllPickups()`. Wyniki są stronicowane (domyślnie 20 na stronę), a dociąganie kolejnych stron obsługuje `loadMoreHandovers()`. Dostępna jest również funkcja odświeżania listy przez `refreshHandovers()`, obsługująca mechanizm pull-to-refresh.

Krok 2 – weryfikacja klienta. Po wybraniu rezerwacji pracownik podaje numer dokumentu tożsamości i kod kraju wydającego. Metoda `confirmCustomerVerification()` waliduje pola (numer nie może być pusty; kod kraju musi mieć dokładnie 2 litery) i wywołuje `apiClientService.startHandover()`. Błędy walidacji są zwracane jako mapa `Map<String, String>` indeksowana po nazwie pola, co pozwala wyświetlić komunikaty przy konkretnych polach formularza.

Krok 3 – inspekcja pojazdu. Pracownik ocenia stan pojazdu: odczyt licznika, poziom paliwa, stan ogumienia i nadwozia (skale liczbowe) oraz ewentualne uszkodzenia opisem tekstowym i zdjęciami. Walidacja `validateInspection()` wymaga podania stanu licznika i – gdy stan ogumienia lub nadwozia jest nieidealny – co najmniej jednego zdjęcia. Błędy inspekcji są przechowywane w dedykowanym polu `inspectionErrors`, odrębnym od ogólnych błędów API. Zdjęcia są przechwytywane przez `PhotoProvider`.

Krok 4 – podgląd kontraktu. Po zatwierdzeniu inspekcji metoda `confirmInspectionAndBuildContract()` wysyła zebrane dane (stan pojazdu, zdjęcia jako multipart/form-data) do API przez `apiClientService.prepareHandover()`. API zwraca URL wygenerowanego dokumentu PDF, który jest buforowany w `contractUrlCache`, jak przedstawiono na *Listing 61*.

Listing 61 Pamięć podręczna URL kontraktu PDF indeksowana po identyfikatorze rezerwacji. Źródło: opracowanie własne.

```
// CarHandoverViewModel.kt (fragment)
/** Cache: reservationId -> contractDocumentUrl (in-memory, cleared on
complete) */
private val contractUrlCache = mutableMapOf<Long, String>()
```

Ekran `HandoverContractScreen` wyświetla dokument PDF przez `PdfViewer` i udostępnia trzy akcje: przejście do podpisu (`proceedToSignature()`), powrót do poprzedniego ekranu oraz ponowną inspekcję pojazdu (`reinspectVehicle()`). Buforowanie zapobiega ponownemu pobieraniu URL przy nawigacji wstecz.

Przy wznowieniu przerwanego procesu (`pickupStatus == STARTED`) `startHandover()` pomija ekran weryfikacji i przechodzi bezpośrednio do inspekcji, jak przedstawiono na *Listing 62*.

Listing 62 Wybór punktu wejścia do procesu wydania na podstawie statusu rezerwacji – wznowienie pomija weryfikację klienta. Źródło: opracowanie własne.

```
fun startHandover(reservation: Reservation) {
    val cachedUrl = contractUrlCache[reservation.id]
    update { it.copy(selectedReservation = reservation, contractHtml =
cachedUrl, ...) }
    viewModelScope.launch {
        when (reservation.pickupStatus) {
```

```
            VehiclePickupStatus.STARTED ->

_effects.emit(CarHandoverEffect.ToInspection(reservation.id))
            else ->
                _effects.emit(CarHandoverEffect.ToVerify(reservation.id))
        }
    }
}
```

Krok 5 – podpis klienta. Przed wyświetleniem pola podpisu system weryfikuje, czy pracownik ma zapisany własny podpis w profilu (`apiClientService.getEmployeeSignature()`). Brak podpisu pracownika (odpowiedź 404) powoduje ustawienie flagi `showMissingSignatureDialog = true` i wyświetlenie okna dialogowego z możliwością przejścia do modułu profilu przez emisję efektu `CarHandoverEffect.ToProfile`. Pracownik może zamknąć dialog (`dismissMissingSignatureDialog()`) i uzupełnić podpis w profilu, po czym wrócić do procesu wydania. Podpis klienta jest rysowany przez `SignatureComponent` oparty o bibliotekę compose-signature v1.0.1.

Krok 6 – zakończenie. Metoda `completeHandover()` wysyła podpis (PNG, ByteArray) do API. Po sukcesie rezerwacja jest usuwana z lokalnej listy, a `contractUrlCache` jest czyszczony dla danego `reservationId`. Emitowany jest efekt `ToCompleted` i wyświetlany ekran potwierdzenia.

7.3.4.3. Zwrot pojazdu (UC-OPE-02)

Proces zwrotu pojazdu, realizujący przypadek użycia UC-OPE-02, jest symetryczny strukturalnie względem procesu wydania i zarządzany przez `CarReturnViewModel`. Stan całego procesu jest przechowywany w klasie `CarReturnUiState`, jak przedstawiono na *Listing 63*.

Listing 63 Niemutowalna klasa stanu procesu zwrotu pojazdu obejmująca wszystkie etapy wielokrokowego wizarda. Źródło: opracowanie własne.

```
data class CarReturnUiState(
    val isLoading: Boolean = false,
    val isRefreshing: Boolean = false,
    val isLoadingMore: Boolean = false,
    val errorMessage: String? = null,
    val isSignatureFieldEnabled: Boolean = false,
    val pendingReturns: List<VehicleReturnListItem> = emptyList(),

    val currentPage: Int = 1,
    val hasNextPage: Boolean = false,

    val selectedReturn: VehicleReturnListItem? = null,

    val odometerKm: String = "",
    val fuelLevel: Int = 3,
    val tireCondition: Int = 10,
    val bodyCondition: Int = 5,
    val photos: List<PhotoItem> = emptyList(),
    val remarks: String = "",

    val formErrors: Map<String, String> = emptyMap(),

    val protocolPreviewHtml: String? = null,
    val signatureBitmapSaved: Boolean = false,
    val showMissingSignatureDialog: Boolean = false
)
```

Zdarzenia nawigacyjne są emitowane jako `SharedFlow<CarReturnEffect>`, analogicznie do procesu wydania, jak przedstawiono na *Listing 64*.

Listing 64 Zdarzenia nawigacyjne procesu zwrotu pojazdu emitowane jako SharedFlow dla jednokrotnego przetworzenia. Źródło: opracowanie własne.

```
sealed interface CarReturnEffect {
    data class ToInspection(val returnId: Int) : CarReturnEffect
    data class ToProtocol(val returnId: Int)   : CarReturnEffect
    data class ToSignature(val returnId: Int)  : CarReturnEffect
    data object ToCompleted : CarReturnEffect
    data object BackToList  : CarReturnEffect
    data object ToProfile   : CarReturnEffect
}
```

Krok 1 – lista aktywnych wynajmów. `CarReturnViewModel` pobiera przy inicjalizacji listę zwrotów oczekujących na realizację przez `apiClientService.getAllReturns()`. Wyniki są stronicowane (20 na stronę), a dociąganie kolejnych stron obsługuje `loadMoreReturns()`. Funkcja `refreshReturns()` obsługuje mechanizm *pull-to-refresh*.

Krok 2 – inspekcja pojazdu. Pracownik ocenia stan pojazdu po zakończeniu wynajmu: stan licznika, poziom paliwa, stan ogumienia i nadwozia oraz ewentualne uwagi. Walidacja `validateForm()` wymaga podania odczytu licznika i – gdy zarejestrowano nieprawidłowości ogumienia lub nadwozia – co najmniej jednego zdjęcia, jak przedstawiono na *Listing 65*.

Listing 65 Walidacja formularza inspekcji zwrotu – wymaga zdjęć przy stwierdzonych nieprawidłowościach stanu pojazdu. Źródło: opracowanie własne.

```
private fun validateForm(state: CarReturnUiState): Map<String, String> {
    val errors = mutableMapOf<String, String>()
    val km = state.odometerKm.toIntOrNull()
    if (km == null || km < 0) errors["odometerKm"] = "Podaj poprawny stan
licznika (>= 0)."

    val requiresPhotos = state.tireCondition < 10 || state.bodyCondition
< 5
    if (requiresPhotos && state.photos.isEmpty()) {
        errors["photos"] = "Dodaj zdjęcia przy stwierdzonych
nieprawidłowościach."
    }
    return errors
}
```

Krok 3 – protokół zwrotu. Metoda `confirmInspectionAndBuildProtocol()` wysyła zebrane dane (stan pojazdu, zdjęcia jako multipart/form-data) do API przez `apiClientService.prepareReturn()`. API zwraca URL wygenerowanego protokołu zwrotu w formacie PDF. URL jest buforowany w `protocolUrlCache`, jak przedstawiono na *Listing 66*.

Listing 66 Pamięć podręczna URL protokołu zwrotu PDF indeksowana po identyfikatorze zwrotu. Źródło: opracowanie własne.

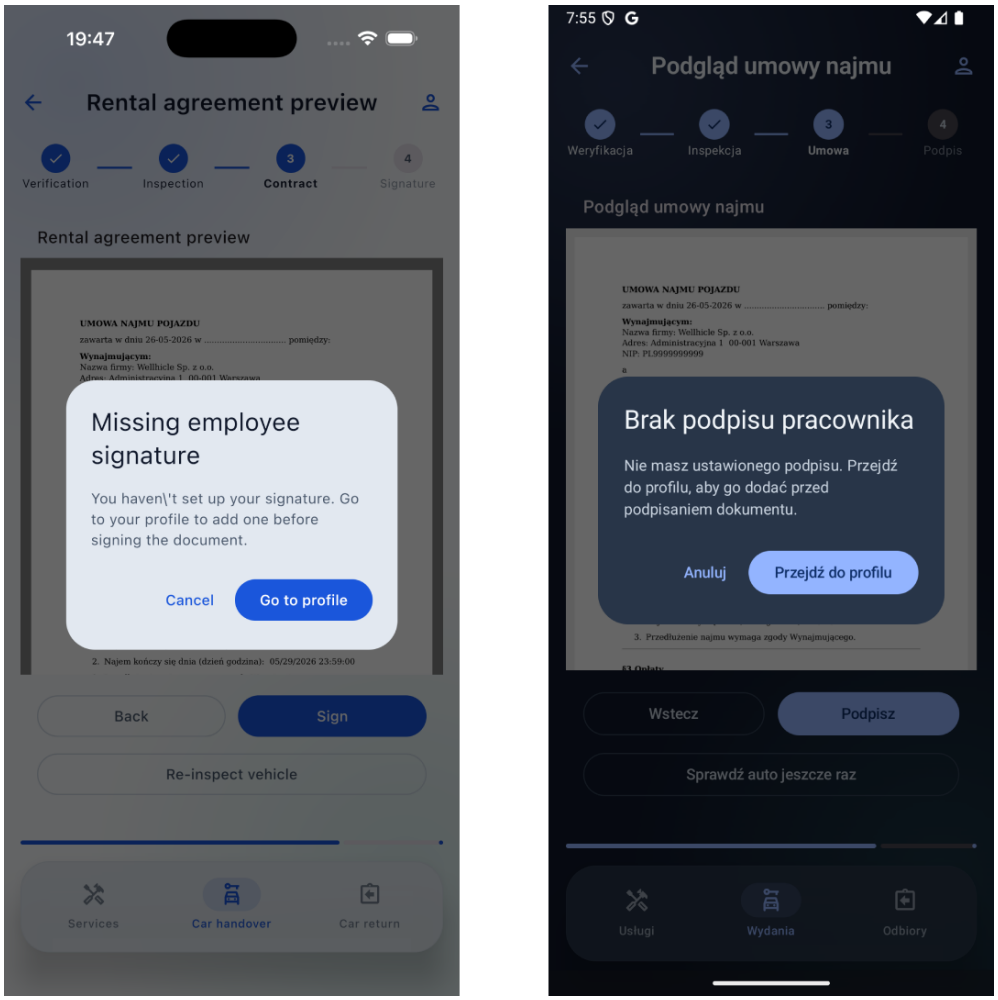
```
// CarReturnViewModel.kt
/** Cache: returnId -> protocolDocumentUrl (in-memory, cleared on
complete) */
private val protocolUrlCache = mutableMapOf<Int, String>()

// po udanym prepareReturn:
protocolUrlCache[r.id] = result.documentUrl
```

Ekran `ReturnProtocolScreen` wyświetla dokument przez `PdfViewer` i udostępnia akcje: przejście do podpisu oraz ponowną inspekcję (`reinspectVehicle()`). Buforowany URL pozwala na wyświetlenie podglądu protokołu bez powtórnego pobierania dokumentu. Jeśli przy

wznawianiu procesu (`returnStatus == PREPARED`) URL jest już w pamięci podręcznej, system przechodzi bezpośrednio do ekranu podglądu protokołu, pomijając ponowną inspekcję.

Krok 4 – podpis klienta. Analogicznie do procesu wydania, przed wyświetleniem pola podpisu `proceedToSignature()` weryfikuje, czy pracownik posiada zapisany podpis w profilu (`apiClientService.getEmployeeSignature()`). Brak podpisu pracownika (odpowiedź 404) wyświetla dialog z opcją przejścia do profilu przez `goToProfileToSetSignature()`, emitującą efekt `CarReturnEffect.ToProfile`, jak przedstawiono na *Rysunek 153*. Podpis klienta jest zbierany przez `SignatureComponent`.



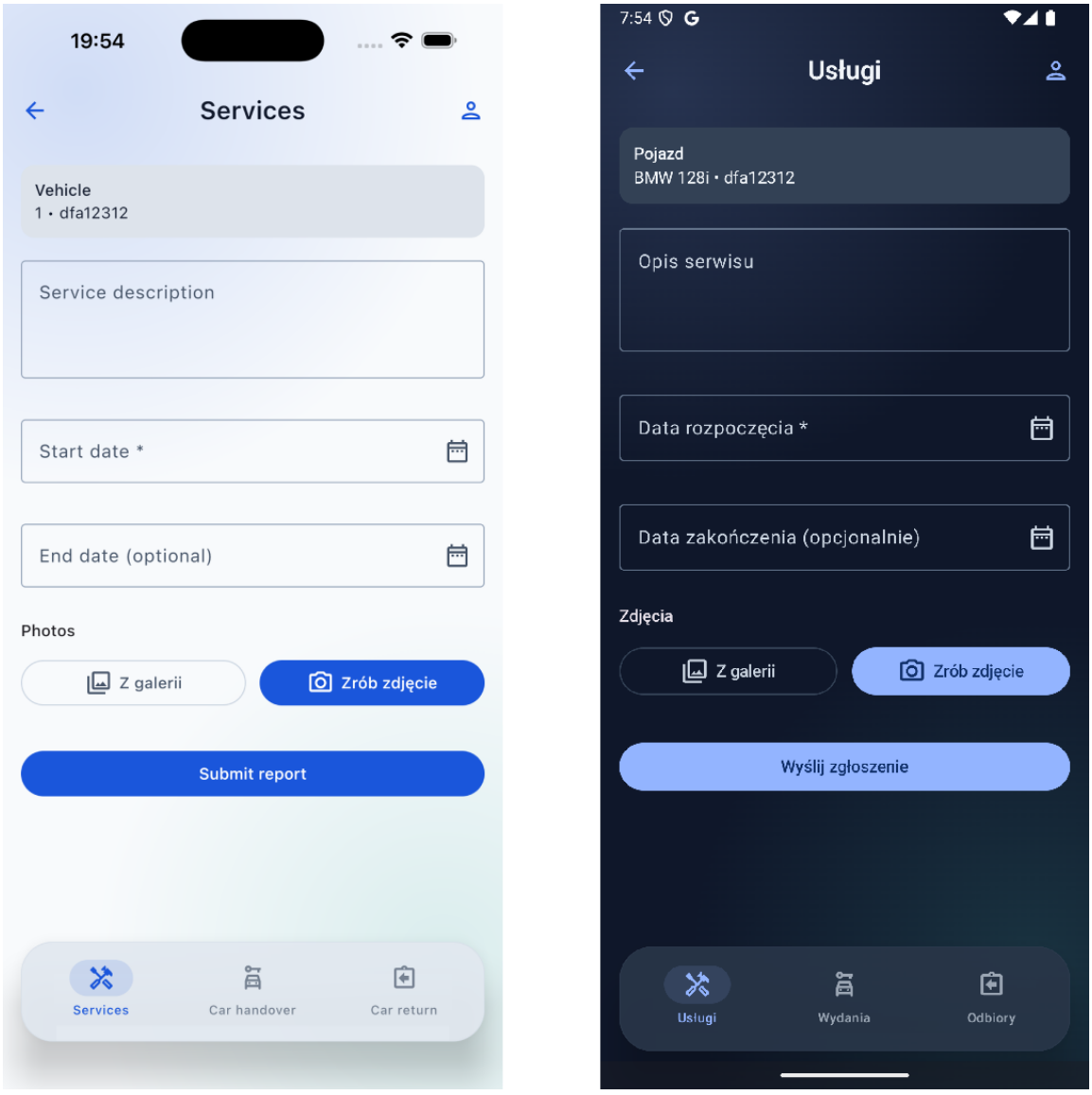
Rysunek 153. Dialog informujący o braku podpisu pracownika z opcją przejścia do ekranu profilu. Źródło: opracowanie własne.

Krok 5 – zakończenie. Metoda `completeReturn()` wysyła podpis (PNG, ByteArray) do API przez `apiClientService.completeReturn()`. Po sukcesie pozycja jest usuwana z lokalnej listy, a `protocolUrlCache` jest czyszczony dla danego *returnId*. Emitowany jest efekt `ToCompleted` i wyświetlany ekran potwierdzenia.

W przypadku błędu sieciowego dane są zapisywane lokalnie przez moduł `core:files` na potrzeby ponownej próby wysłania.

7.3.4.4. Zgłaszanie usterek (UC-OPE-03)

Moduł `feature:services` pozwala na zgłaszanie usterek do systemu niezależnie od procesów wydania i zwrotu. Przepływ składa się z trzech ekranów: lista pojazdów floty (`VehiclesListScreen`), szczegóły pojazdu (`VehicleDetailScreen`) oraz formularz zgłoszenia usterki (`ServiceReportScreen`). Zgłoszenie jest serializowane jako `VehicleIssueReportRequest` i wysyłane do API za pośrednictwem `ServiceIssuesViewModel`. Ekran formularza przedstawiono na *Rysunek 154*.



Rysunek 154. Ekran zgłaszania usterki pojazdu. Źródło: opracowanie własne.

7.3.5. Ciekawe rozwiązania

W tym punkcie omówiono wybrane rozwiązania techniczne, które wykraczają poza standardowe użycie wykorzystanych bibliotek i wynikają z wieloplatformowego charakteru projektu. Każde z nich stanowi odpowiedź na konkretne ograniczenie platformy lub wymaganie procesów obsługiwanych przez aplikację.

7.3.5.1. Mechanizm expect/actual – wzorzec interfejsu platformowego

Kotlin Multiplatform udostępnia dwa sposoby implementacji kodu platformo-specyficznego: słowa kluczowe *expect/actual* oraz zwykłe interfejsy Kotlin z implementacjami w katalogach źródłowych *androidMain/iosMain*. W projekcie Wellhicle zastosowano oba podejścia, zależnie od sytuacji.

Dla *PhotoProvider* – komponentu odpowiedzialnego za przechwytywanie zdjęć aparatem – zastosowano wzorzec interfejsu. W *commonMain* zdefiniowano interfejs *PhotoProvider*, a *composable factory* jest zadeklarowana jako *expect*, jak przedstawiono na *Listing 67*.

Listing 67 Interfejs komponentu aparatu/galerii z asynchronicznymi metodami zwracającymi null przy anulowaniu. Źródło: opracowanie własne.

```
// commonMain - PhotoProvider.kt
interface PhotoProvider {
    suspend fun pickFromGallery(): PickedImage?
    suspend fun takePhoto(): PickedImage?
}

@Composable
expect fun rememberPhotoProvider(): PhotoProvider
```

Implementacja Android w *androidMain* korzysta z *ActivityResultLauncher*, *FileProvider* i mechanizmu coroutine *suspendCancellableCoroutine* do zawieszenia wywołującego coroutine do momentu uzyskania wyniku z launchera systemowego, jak przedstawiono na *Listing 68*.

Listing 68 Implementacja aparatu na Androidzie zawieszająca coroutine do powrotu wyniku z ActivityResultLauncher . Źródło: opracowanie własne.

```
// androidMain - PhotoProvider.android.kt (fragment)
@Composable
actual fun rememberPhotoProvider(): PhotoProvider {
    val context = LocalContext.current
    var cameraCont by remember { mutableStateOf<(PickedImage?) ->
Unit>({}) }
    var pendingCameraUri by remember { mutableStateOf<Uri?>(null) }

    val takePictureLauncher = rememberLauncherForActivityResult(
        ActivityResultContracts.TakePicture()
    ) { success: Boolean ->
        val uri = pendingCameraUri
        val result = if (success && uri != null)
context.readPickedImage(uri) else null
        cameraCont(result)
    }

    return remember {
        object : PhotoProvider {
            override suspend fun takePhoto(): PickedImage? =
                suspendCancellableCoroutine { cont ->
                    cameraCont = { cont.resume(it) }
                    val uri = context.createTempCameraImageUri()
                    pendingCameraUri = uri
                    takePictureLauncher.launch(uri)
                }
        }
    }
}
```

7.3.5.2. LockLandscapeEffect – blokada orientacji ekranu

Dla `LockLandscapeEffect` – efektu blokującego orientację ekranu podczas zbierania podpisu – zastosowano mechanizm `expect/actual`, jak przedstawiono na *Listing 69*.

Listing 69 Deklaracja efektu blokady orientacji poziomej z pustą implementacją po stronie iOS. Źródło: opracowanie własne.

```
// commonMain - LockLandscapeEffect.kt
@Composable
expect fun LockLandscapeEffect()
```

Implementacja Android (w `DisposableEffect`) ustawia `requestedOrientation = SCREEN_ORIENTATION_LANDSCAPE` i przywraca oryginalną wartość po opuszczeniu kompozycji, jak przedstawiono na Listing 70.

Listing 70 Blokada orientacji poziomej ekranu na czas zbierania podpisu z przywróceniem oryginału po opuszczeniu ekranu. Źródło: opracowanie własne.

```
// androidMain - LockLandscapeEffect.android.kt
@Composable
actual fun LockLandscapeEffect() {
    val context = LocalContext.current
    DisposableEffect(Unit) {
        val activity = context as? Activity
            ?: return@DisposableEffect onDispose {}
        val originalOrientation = activity.requestedOrientation
        activity.requestedOrientation =
ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE
        onDispose {
            activity.requestedOrientation = originalOrientation
        }
    }
}
```

Implementacja iOS jest pusta – blokada orientacji na iOS wymaga ingerencji w cykl życia `UIViewController` poza możliwościami Compose Multiplatform w bieżącej wersji.

Blokada orientacji poziomej na ekranie podpisu zwiększa powierzchnię pola podpisu, co przekłada się na dokładność złożonego podpisu cyfrowego przez klienta.

7.3.5.3. WithAppLocale – wielojęzyczność bez restartu aplikacji

Aplikacja obsługuje dwa języki: polski i angielski. Zasoby tekstowe są zorganizowane per moduł w plikach *composeResources/values/strings.xml* (język domyślny) i *values-en/strings.xml*. Zmiana języka odbywa się bez restartu aplikacji za pomocą *composable* `WithAppLocale`. Na Androidzie `WithAppLocale` używa `CompositionLocalProvider` do nadpisania konfiguracji `Locale` dla całego drzewa `composable`, jak przedstawiono na *Listing 71*.

Listing 71 Nadpisanie konfiguracji lokalizacji dla drzewa composable bez restartu aktywności na Androidzie. Źródło: opracowanie własne.

```
// androidMain - WithAppLocale.android.kt
@Composable
actual fun WithAppLocale(languageCode: String, content: @Composable () ->
Unit) {
    val currentConfig = LocalConfiguration.current
    val overriddenConfig = Configuration(currentConfig).apply {
        setLocale(Locale(languageCode))
    }
    CompositionLocalProvider(LocalConfiguration provides
overriddenConfig) {
```

```
        content()
    }
}
```

Na iOS ustawienie języka odbywa się przez `NSUserDefaults.standardUserDefaults["AppleLanguages"]`, a *Compose Resources* odczytuje je z ustawień systemowych. Implementacja `WithAppLocale` po stronie iOS jest zatem pusta, jak przedstawiono na *Listing 72*.

Listing 72 Pusta implementacja zmiany języka przez composable na iOS – deleguje do NSUserDefaults w repozytorium. Źródło: opracowanie własne.

```
// iosMain - WithAppLocale.ios.kt
@Composable
actual fun WithAppLocale(languageCode: String, content: @Composable () ->
Unit) {
    // Na iOS ustawienie języka następuje przez NSUserDefaults w
UserProfileRepository.
    content()
}
```

Zmiana języka na iOS jest obsługiwana przez metodę przedstawioną na *Listing 73*.

Listing 73 Ustawienie preferowanego języka aplikacji w NSUserDefaults na iOS. Źródło: opracowanie własne.

```
// iosMain - UserProfileRepository.ios.kt
actual fun changeLanguage(language: String) {
    NSUserDefaults.standardUserDefaults
        .setObject(arrayListOf(language), "AppleLanguages")
}
```

Preferencja językowa jest przechowywana w UserSettingsStorage (opartym o *multiplatform-settings*), co zapewnia zachowanie wybranego języka między sesjami.

7.3.5.4. PdfViewer – podgląd dokumentu umowy na dwóch platformach

Wyświetlanie dokumentu umowy najmu i protokołu zwrotu w formacie PDF wymaga platformo-specyficznych rozwiązań. Deklaracja w *commonMain* jest zwykłym *expect composable*:

Listing 74 Deklaracja expect komponentu wyświetlającego dokumenty PDF wymagająca implementacji per platforma. Źródło: opracowanie własne.

```
// commonMain - PdfViewer.kt
@Composable
expect fun PdfViewer(url: String, modifier: Modifier = Modifier)
```

Na Androidzie `PdfViewer` pobiera plik do katalogu cache urządzenia, a następnie renderuje strony przez `PdfRenderer` z pakietu android.graphics.pdf. Każda strona jest konwertowana do Bitmap i wyświetlana jako element `LazyColumn`. Plik jest buforowany – przy ponownym otwarciu ekranu podglądu nie jest pobierany ponownie, o ile istnieje w cache, jak przedstawiono na *Listing 75*.

Listing 75 Renderowanie stron PDF do Bitmap z lokalnym buforowaniem pliku na Androidzie. Źródło: opracowanie własne.

```
// androidMain - PdfViewer.android.kt (fragment)
val cacheFile = File(context.cacheDir, "pdf_${url.hashCode()}.pdf")
if (!cacheFile.exists() || cacheFile.length() == 0L) {
    // pobierz z sieci
}
val renderer = PdfRenderer(ParcelFileDescriptor.open(cacheFile,
MODE_READ_ONLY))
val pages = (0 until renderer.pageCount).map { i ->
    renderer.openPage(i).use { page ->
        val bitmap = Bitmap.createBitmap(page.width * 2, page.height * 2,
ARGB_8888)
```

```
        page.render(bitmap, null, null, RENDER_MODE_FOR_DISPLAY)
        bitmap
    }
}
```

Na iOS, gdzie nie ma bezpośredniego odpowiednika `PdfRenderer` dostępnego przez Kotlin/Native, zastosowano `WKWebView` osadzone w `composable` przez mechanizm `UIKitView` z Compose Multiplatform. `WKWebView` wyświetla pliki PDF przez standardowe żądanie HTTP, co eliminuje konieczność implementacji własnego `renderera`. Stany ładowania i błędu są obsługiwane przez delegat nawigacji `WKNavigationDelegateProtocol`, jak przedstawiono na *Listing 76*.

Listing 76 Wyświetlanie dokumentu PDF przez WKWebView osadzone w composable za pomocą UIKitView na iOS. Źródło: opracowanie własne.

```
// iosMain - PdfViewer.ios.kt (fragment)
@OptIn(ExperimentalForeignApi::class)
@Composable
actual fun PdfViewer(url: String, modifier: Modifier) {
    UIKitView(
        factory = {
            val webView = WKWebView()
            webView.navigationDelegate = object : NSObject(),
                    WKNavigationDelegateProtocol {
                override fun webView(webView: WKWebView,
                        didFinishNavigation: WKNavigation?) {
                    loadState = IosLoadState.Success
                }
                override fun webView(webView: WKWebView,
                        didFailNavigation: WKNavigation?, withError:
NSError) {
                    loadState = IosLoadState.Error
                }
            }
            val nsUrl = NSURL.URLWithString(url)
            if (nsUrl != null)
webView.loadRequest(NSURLRequest.requestWithURL(nsUrl))
            webView
        },
        modifier = Modifier.fillMaxSize()
    )
}
```

7.3.5.5. Buforowanie URL dokumentu w pamięci operacyjnej

Zarówno `CarHandoverViewModel`, jak i `CarReturnViewModel` przechowują mapę URL-ów buforowanych w pamięci operacyjnej, indeksowaną po identyfikatorze procesu. Mechanizm ten rozwiązuje dwa problemy jednocześnie.

Po pierwsze, URL dokumentu PDF jest pobierany z API jednorazowo podczas generowania kontraktu lub protokołu i buforowany w pamięci. Kolejne wywołania (np. powrót do ekranu podglądu po nawigacji wstecz) zwracają wartość z pamięci podręcznej bez dodatkowego żądania sieciowego. Jest to istotne w warunkach słabego zasięgu sieciowego typowych dla hal i parkingów.

Po drugie, przy wznowieniu przerwanego procesu system sprawdza pamięć podręczną przed wybraniem punktu wejścia, jak przedstawiono na *Listing 77*.

Listing 77 Wybór punktu wejścia przy wznawianiu zwrotu – pomija inspekcję gdy protokół jest dostępny w pamięci podręcznej. Źródło: opracowanie własne.

```
// CarReturnViewModel.kt (fragment startReturn)
fun startReturn(item: VehicleReturnListItem) {
    val cachedUrl = protocolUrlCache[item.id]
    update { it.copy(selectedReturn = item, protocolPreviewHtml =
cachedUrl, ...) }
    viewModelScope.launch {
        when (item.returnStatus) {
            VehicleReturnStatus.PREPARED -> {
                if (cachedUrl != null) {
                    _effects.emit(CarReturnEffect.ToProtocol(item.id))
                } else {
                    _effects.emit(CarReturnEffect.ToInspection(item.id))
                }
            }
            else -> _effects.emit(CarReturnEffect.ToInspection(item.id))
        }
    }
}
```

Jeśli dokument jest dostępny w pamięci podręcznej i pojazd ma status *PREPARED*, system kieruje pracownika bezpośrednio do ekranu protokołu, pomijając ponowną inspekcję. Po zakończeniu procesu wpis jest usuwany, co zapobiega wyciekom pamięci, jak przedstawiono na *Listing 78*.

Listing 78 Usunięcie URL protokołu z pamięci podręcznej po zakończeniu procesu zwrotu. Źródło: opracowanie własne.

```
protocolUrlCache.remove(r.id)
```

7.4. Wdrażanie i monitorowanie

W niniejszym podrozdziale opisano sposób uruchomienia platformy Wellhicle oraz proces dostarczania kolejnych wersji jej komponentów. System składa się z kilku niezależnie wdrażanych aplikacji przeglądarkowych, serwera aplikacyjnego, usług pomocniczych oraz wspólnej infrastruktury, dlatego proces wdrożenia musiał zapewniać powtarzalność, kontrolę wersji i możliwość szybkiej diagnostyki po publikacji zmian.

Opis obejmuje środowisko zespołu projektowego, dlatego skupia się na rozwiązaniach użytych podczas realizacji pracy: klastrze Kubernetes uruchomionym w wariancie k3s, rejestrze obrazów Harbor, mechanizmach GitHub Actions oraz stosie monitorowania opartym na Grafanie, Prometheusie, Loki i Tempo.

7.4.1. Konfiguracja środowiska uruchomieniowego

Środowisko uruchomieniowe platformy Wellhicle oparto na lokalnym klastrze Kubernetes uruchomionym w dystrybucji k3s. Zgodnie z architekturą systemu, cały ruch zewnętrzny przechodzi przez serwer *proxy*, który stanowi jedyny punkt wejścia do systemu. W warstwie wdrożeniowej dostęp z internetu realizowany jest za pośrednictwem Cloudflare Tunnel, a następnie kierowany do serwera NGINX pełniącego funkcję serwera *proxy* [74]. Dopiero ta warstwa rozdziela żądania pomiędzy serwery WWW udostępniające aplikacje przeglądarkowe oraz serwer aplikacyjny obsługujący API.

Adresy publiczne przypisano zarówno do aplikacji użytkowych, jak i wybranych narzędzi środowiska wdrożeniowego. Dla użytkowników istotne są adresy aplikacji przeglądarkowych oraz API serwera aplikacyjnego. Adresy takie jak grafana.wellhicle.com, harbor.wellhicle.com i files.wellhicle.com pełnią funkcję techniczną: służą odpowiednio do obserwacji systemu, publikowania

obrazów kontenerowych oraz obsługi serwera plików. Mapowanie adresów na przeznaczenie usług przedstawiono w *Tabela 59*.

Tabela 59 Zestawienie adresów dostępnych w systemie Wellhicle oraz ich przeznaczenie. Źródło: opracowanie własne.

Adres	**Przeznaczenie**
api.wellhicle.com	API serwera aplikacyjnego używane przez aplikację mobilną oraz system ERP, z pominięciem serwerów WWW
dashboard.wellhicle.com	Serwer WWW aplikacji do zarządzania wypożyczalnią
admin.wellhicle.com	Serwer WWW aplikacji do administrowania platformą
{nazwa}.wellhicle.com	Dynamiczna subdomena klienta kierowana do aplikacji korzystania z usług wypożyczalni
wellhicle.com	Serwer WWW aplikacji prezentującej ofertę systemu
admin-dev.wellhicle.com	Deweloperska wersja serwera WWW aplikacji administracyjnej
dashboard-dev.wellhicle.com	Deweloperska wersja serwera WWW aplikacji do zarządzania wypożyczalnią
api-dev.wellhicle.com	Deweloperska wersja API serwera aplikacyjnego
harbor.wellhicle.com	Adres techniczny prywatnego rejestru obrazów kontenerowych Harbor
files.wellhicle.com	Adres techniczny serwera plików RustFS; obsługuje zasoby publiczne i prywatne zgodnie z regułami dostępu
grafana.wellhicle.com	Adres techniczny panelu monitorowania Grafana

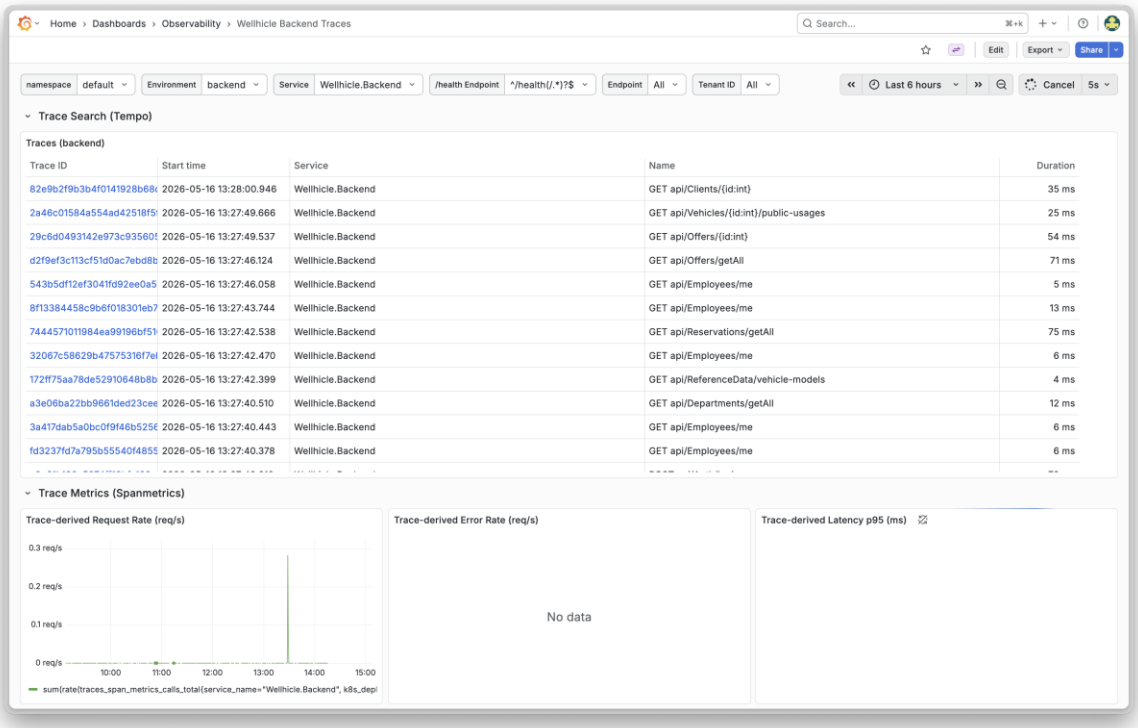
Model *same-origin* opisany w podrozdziale *6.2* został zachowany przez obsługę ruchu w serwerze proxy. Aplikacje przeglądarkowe i serwer aplikacyjny są udostępniane w taki sposób, aby przeglądarka traktowała komunikację z API jako ruch w obrębie tego samego źródła. Rozróżnienie komponentów odbywa się po domenie i ścieżce żądania, co ogranicza potrzebę dodatkowej konfiguracji *CORS* oraz upraszcza komunikację między aplikacją przeglądarkową a serwerem aplikacyjnym.

W środowisku uruchomiono wspólne komponenty infrastrukturalne odpowiedzialne za przechowywanie obrazów kontenerowych, obsługę plików, zbieranie danych diagnostycznych oraz wizualizację stanu systemu. Ich zadaniem było zapewnienie podstawowych usług technicznych, z których korzystały serwer aplikacyjny, serwery WWW oraz usługi pomocnicze. W środowisku uruchomiono następujące komponenty:

- **Harbor** – prywatny rejestr obrazów kontenerowych serwera aplikacyjnego, serwerów WWW oraz usług pomocniczych, używany przez przepływy wdrożeniowe oraz zasoby Kubernetes.
- **RustFS** – wdrożeniowa realizacja serwera plików, zgodna z interfejsem S3. Serwer plików rozdziela zasoby publiczne, takie jak zdjęcia pojazdów, oraz zasoby prywatne, na przykład dokumenty dostępne po autoryzacji lub po wygenerowaniu tymczasowego klucza dostępu.
- **OpenTelemetry Collector** – komponent zbierający dane telemetryczne z aplikacji i przekazujący je do narzędzi obserwacyjnych.
- **Loki** – system agregacji logów z kontenerów i usług działających w klastrze.
- **Prometheus** – system zbierania metryk wykorzystywany do obserwacji stanu usług i infrastruktury.
- **Tempo** – system przechowywania śladów rozproszonych, umożliwiający analizę przebiegu żądań między usługami.

- **Grafana** – panel wizualizacji metryk, logów i śladów, stanowiący główny interfejs do obserwacji działania systemu.

Stos monitorowania był wykorzystywany podczas testowania systemu, sprawdzania skutków wdrożeń oraz diagnozowania problemów z wydajnością i dostępnością usług. Na *Rysunek 155* przedstawiono widok panelu Grafany, w którym zestawiono dane z obszaru metryk i śladów aplikacyjnych.



Rysunek 155 Widok panelu ze śladami aplikacyjnymi oraz metrykami z Grafany. Źródło: opracowanie własne.

W klastrze skonfigurowano również mechanizm uruchamiania agentów GitHub Actions oraz proces nasłuchujący odbierający zdarzenia z platformy GitHub. Dzięki temu przepływy integracji i wdrażania kodu mogły być wykonywane wewnątrz tej samej infrastruktury, w której działały rejestr Harbor oraz docelowe zasoby Kubernetes. Agentom nadano uprawnienia przez konto serwisowe i mechanizmy RBAC, dlatego zakres ich dostępu mógł zostać ograniczony do operacji wymaganych przez proces wdrożenia. Fragment tej konfiguracji przedstawiono w *Listing 79*.

Listing 79 Fragment konfiguracji konta serwisowego używanego do wdrażania aplikacji w klastrze Kubernetes. Źródło: opracowanie własne.

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: github-runner-deployer
  namespace: default
  annotations:
    description: "ServiceAccount used by GitHub Actions runners to deploy
applications"
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: github-runner-deployer
rules:
```

```
  - apiGroups: ["apps"]
    resources: ["deployments", "replicasets", "statefulsets",
"daemonsets"]
    verbs: ["get", "list", "watch", "create", "update", "patch",
"delete"]
  - apiGroups: [""]
    resources: ["pods", "pods/log", "services", "configmaps", "secrets"]
    verbs: ["get", "list", "watch", "create", "update", "patch",
"delete"]
...
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: github-runner-deployer
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: github-runner-deployer
subjects:
- kind: ServiceAccount
  name: github-runner-deployer
  namespace: default
```

Wszystkie wspólne komponenty wdrożono z użyciem wykresów Helm albo manifestów Kubernetes przechowywanych w repozytorium infrastruktury. Pliki Kustomize scalają te elementy w jeden stan docelowy środowiska, dzięki czemu możliwe jest ponowne odtworzenie konfiguracji po awarii węzła lub przeniesieniu środowiska na nowy klaster.

Repozytorium infrastructure pełni rolę źródła stanu docelowego dla wspólnych elementów środowiska. Głównym punktem wejścia jest plik kustomization.yaml, który łączy manifesty usług, konfigurację NGINX oraz przypięte wersje wykresów Helm. W tym pliku znajdują się między innymi definicje wdrożeń Prometheusa, Loki, Tempo, OpenTelemetry Collector, Grafany, Harbora, RustFS i agentów GitHub Actions. Sekrety, takie jak hasła administracyjne, token Cloudflare Tunnel i dane dostępowe do Harbor, nie są przechowywane w repozytorium. Są przekazywane z sekretów GitHub podczas uruchamiania procesu bootstrapu.

Rejestr Harbor nie został wystawiony bezpośrednio w publicznej sieci. Ruch do rejestru jest kierowany przez NGINX na dedykowaną subdomenę harbor.wellhicle.com, natomiast sama usługa Harbor działa w klastrze jako ClusterIP. Oznacza to, że bezpośredni dostęp do niej jest możliwy tylko z sieci Kubernetes. NGINX przekazuje do Harbor także nagłówki informujące o zewnętrznym dostępie przez HTTPS, co jest wymagane do poprawnego generowania adresów w interfejsie rejestru i podczas operacji *docker push*.

Podczas pierwszych prób publikacji obrazów kontenerowych o dużym rozmiarze warstw pojawił się problem odrzucania żądań przez serwer proxy z powodu domyślnego limitu wielkości treści żądania HTTP. W konfiguracji NGINX dla subdomeny rejestru zwiększono parametr `client_max_body_size` do wartości pięciu gigabajtów, co umożliwiło bezproblemowe wykonywanie operacji `docker push` oraz przesyłanie większych plików do magazynu obiektowego.

Dla publicznych stron poszczególnych wypożyczalni skonfigurowano regułę routingu opartą na wzorcu subdomeny {nazwa}.wellhicle.com. Jest to wdrożeniowa realizacja mechanizmu opisanego w podrozdziale *6.2*, w którym dynamiczne subdomeny klientów są przepisywane na ścieżki wewnętrzne odczytywane przez aplikację do korzystania z usług wypożyczalni. Dzięki temu jedno wdrożenie serwera WWW aplikacji do korzystania z wypożyczalni może obsługiwać wiele adresów wypożyczalni bez ręcznego dodawania konfiguracji dla każdego nowego klienta.

W klastrze uruchomiono również instancję Microsoft SQL Server jako bazę danych relacyjną dla serwera aplikacyjnego. Baza jest utrzymywana jako *StatefulSet*, czyli rodzaj zasobu Kubernetes przeznaczony dla aplikacji stanowych, które wymagają trwałego wolumenu danych. Zmiany schematu są stosowane podczas wdrożenia serwera aplikacyjnego przez jednorazowe zadanie Kubernetes uruchamiane przed aktualizacją aplikacji. Pozwala to zachować zgodność wersji kodu z wersją schematu bazy danych.

7.4.2. Proces integracji kodu

Proces integracji kodu miał na celu ograniczenie ryzyka wprowadzenia do głównej gałęzi zmian, które nie budują się poprawnie lub naruszają podstawowe reguły jakości projektu. Dla repozytoriów aplikacyjnych przygotowano przepływy GitHub Actions uruchamiane automatycznie po zgłoszeniu *pull requestu* do gałęzi `main`. *Pull request* oznacza propozycję włączenia zmian z jednej gałęzi kodu do innej, zwykle po sprawdzeniu ich przez zespół oraz automatyczne testy. Dzięki temu weryfikacja zmian następowała przed ich dołączeniem do wersji, z której wykonywano późniejsze wdrożenia.

W aplikacjach *frontendowych* wspólny wzorzec obejmował pobranie kodu, przygotowanie środowiska Node.js, instalację zależności poleceniem `npm ci`, uruchomienie lintera oraz budowę aplikacji. Na *Listing 80* przedstawiono fragment przepływu dla aplikacji do korzystania z usług wypożyczalni. Zastosowanie osobnych przepływów dla aplikacji *frontendowych* pozwalało na niezależny rozwój i sprawdzanie każdej z nich.

Listing 80 Fragment przepływu dla aplikacji do korzystania z usług wypożyczalni Źródło: opracowanie własne.

```yaml
name: PR CI

on:
  pull_request:
    branches:
      - main

jobs:
  build-and-test:
    runs-on: arc-runners
    env:
      GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '22'
          cache: 'npm'
          registry-url: 'https://npm.pkg.github.com'
          scope: '@wellhicle'

      - name: Install dependencies
        run: npm ci

      - name: Run lint
        run: npm run lint

      - name: Build application
        run: npm run build
        env:
```

```
          NODE_ENV: production
```

W repozytorium *backendowym* proces integracji uwzględniał dodatkowe kroki wynikające z technologii .NET. Przed kompilacją uruchamiano skrypt generujący publiczny eksport kodów błędów w formacie JSON, a następnie wykonywano przywrócenie zależności, kompilację rozwiązania oraz testy automatyczne. Taki układ sprawdzał zarówno poprawność samego kodu aplikacji, jak i spójność artefaktów używanych przez pozostałe części systemu.

Aplikacja mobilna korzystała z analogicznej zasady: zmiana była sprawdzana w oddzielnym przepływie *pull requestu*, zanim mogła zostać połączona z główną gałęzią repozytorium.

Wspólną cechą wszystkich tych przepływów było to, że nie wykonywały one wdrożenia produkcyjnego. Ich zadaniem była weryfikacja jakości zmiany oraz szybkie przekazanie informacji zwrotnej autorowi *pull requestu*. Podsumowanie oraz charakterystykę poszczególnych przepływów integracyjnych przedstawiono w *Tabela 60*.

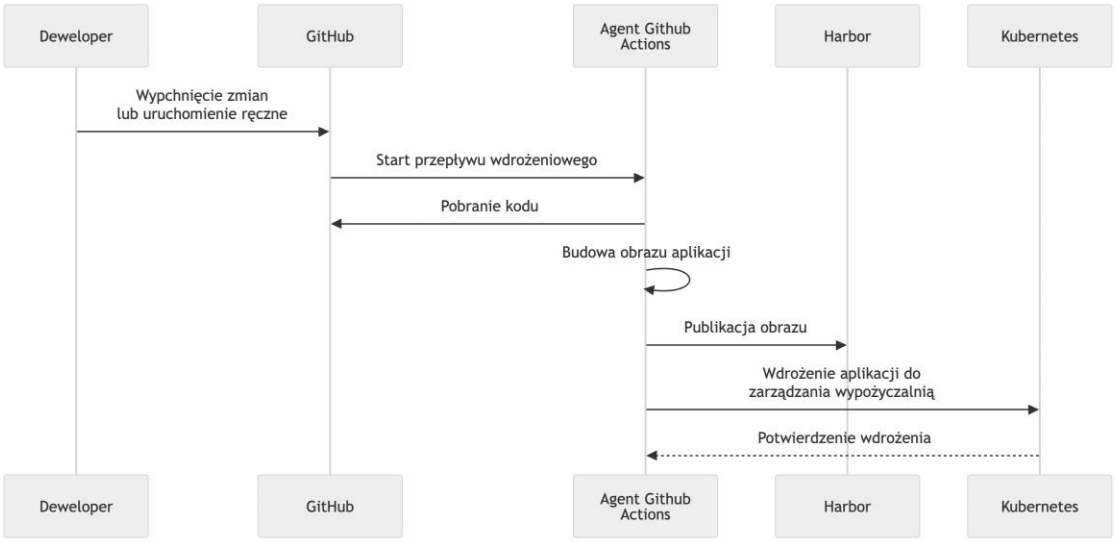
Tabela 60 Zestawienie głównych przepływów integracji kodu. Źródło: opracowanie własne.

Komponent	Zdarzenie wyzwalające	Główne kroki
Aplikacje frontendowe	pull_request do gałęzi main	Instalacja zależności, linter, budowanie aplikacji Angular
Serwer aplikacyjny	pull_request do gałęzi main	Walidacja eksportu kodów błędów, restore, build, testy .NET
Aplikacja moblina	pull_request do gałęzi main	Budowanie i sprawdzenie projektu mobilnego Kotlin

7.4.3. Proces wdrażania kodu

Proces wdrażania rozpoczynał się po połączeniu zmian z gałęzią main albo po ręcznym uruchomieniu przepływu przez *workflow_dispatch*. *Workflow_dispatch* jest zdarzeniem GitHub Actions pozwalającym uruchomić przepływ ręcznie, na przykład z wyborem środowiska lub konkretnego commita. Rozdzielenie integracji kodu od wdrożenia było istotne, ponieważ *pull request* sprawdzał poprawność zmiany, natomiast przepływ wdrożeniowy budował obraz kontenerowy, publikował go w rejestrze Harbor i aktualizował zasoby Kubernetes dla serwerów WWW, serwera aplikacyjnego oraz usług pomocniczych.

Na *Rysunek 156* przedstawiono uproszczony przebieg wdrożenia aplikacji. Zdarzenie w repozytorium GitHub uruchamia agenta w klastrze. Agent pobiera kod, buduje obraz Docker, wysyła go do rejestru Harbor, aktualizuje *tag* obrazu w manifestach Kubernetes, a następnie wykonuje `kubectl apply`. Ostatnim krokiem jest sprawdzenie stanu wdrożenia za pomocą `kubectl rollout status`, co pozwala wykryć nieudane uruchomienie nowych wersji komponentów systemu.



Rysunek 156 Uproszczony przepływ wdrażania aplikacji do zarządzania wypożyczalnią. Źródło: opracowanie własne.

W środowisku deweloperskim wdrożenie aplikacji następowało automatycznie po wypchnięciu zmian do gałęzi main. W przepływach GitHub Actions warunek ten uruchamiał zadania oznaczone jako dev, które budowały obrazy z sufiksem -dev, na przykład backend-dev. Środowisko produkcyjne było uruchamiane ręcznie przez *workflow_dispatch*, co pozwalało wskazać docelowe środowisko oraz opcjonalnie konkretny identyfikator commita. Taki podział ograniczał ryzyko przypadkowej publikacji nieprzetestowanej wersji na produkcji.

Wdrożenie aplikacji przeglądarkowej polegało na zbudowaniu statycznych plików i udostępnieniu ich przez serwer WWW działający w kontenerze NGINX. Pliki Dockerfile używały dwóch etapów: w pierwszym instalowano zależności i budowano aplikację, a w drugim kopiowano wynik budowania do lekkiego obrazu nginx:alpine. W przepływie wdrożeniowym obrazy były tagowane jako latest oraz skrótem commita, co ułatwiało powiązanie działającej wersji z konkretną zmianą w repozytorium. Dokładny sposób definicji obrazu, uwzględniający m.in. pobieranie tokenu z sekretów oraz czyszczenie domyślnych katalogów NGINX, ilustruje *Listing 81*.

Listing 81 Treść pliku Dockerfile dla aplikacji frontendowych Źródło: opracowanie własne.

```
FROM node:22-alpine AS build
WORKDIR /app
COPY . .
RUN --mount=type=secret,id=npmroot \
    GITHUB_TOKEN=$(cat /run/secrets/npmroot) npm install
RUN npm run build
FROM nginx:alpine
RUN rm -rf /etc/nginx/conf.d/* /usr/share/nginx/html/*
COPY nginx.conf /etc/nginx/nginx.conf
COPY --from=build /app/dist/customer-frontend/browser
/usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Konfiguracja serwerów WWW rozróżniała środowiska przez argumenty przekazywane podczas budowania obrazu. Dla środowiska deweloperskiego ustawiano nazwę usługi serwera aplikacyjnego backend-dev, natomiast dla produkcji nazwę backend. Dzięki temu aplikacje przeglądarkowe uruchamiane w klastrze odwoływały się do właściwego backend, a nie do adresu wpisanego na stałe w kodzie.

Wdrożenie serwera aplikacyjnego było bardziej rozbudowane niż wdrożenie serwerów WWW, ponieważ obejmowało kilka obrazów kontenerowych oraz migrację bazy danych. Przepływ budował i publikował obraz serwera aplikacyjnego, obraz procesu roboczego EmailWorker oraz obraz usługi

LibreOffice. EmailWorker odpowiada za obsługę zadań wysyłki wiadomości e-mail przekazywanych asynchronicznie przez broker wiadomości, zgodnie z podziałem opisanym w podrozdziale *6.2*. Usługa LibreOffice jest wdrożeniową częścią generatora dokumentów, używaną przy tworzeniu plików PDF na podstawie szablonów. Następnie aktualizowano manifesty Kubernetes, stosowano pliki *ConfigMap*, uruchamiano zadanie migracji bazy danych i dopiero po jego zakończeniu wdrażano serwer aplikacyjny oraz usługi pomocnicze. *ConfigMap* to zasób Kubernetes używany do przechowywania konfiguracji aplikacji poza obrazem kontenera, na przykład nazw usług lub ustawień środowiskowych. Sposób konfiguracji tego kroku w przepływie wdrożeniowym przedstawiono w *Listing 82*. Taka kolejność realizuje zasadę z podrozdziału *6.2*, według której schemat bazy danych jest synchronizowany z wersją wdrożonego kodu podczas uruchamiania nowej wersji serwera aplikacyjnego.

Listing 82 Fragment przepływu wdrożeniowego uruchamiający migracje bazy danych. Źródło: opracowanie własne.

```
- name: Run database migrations (prod)
  run: |
    kubectl delete job backend-migration --ignore-not-found
    kubectl apply -f k8s/migration-job.yaml
    kubectl wait --for=condition=complete job/backend-migration --
timeout=5m
```

7.4.4. Ciekawe rozwiązania

Pierwszym rozwiązaniem wartym wyróżnienia było uruchomienie agentów GitHub Actions bezpośrednio w klastrze Kubernetes. Dzięki temu proces budowania i wdrażania mógł korzystać z tej samej sieci, w której działały Harbor oraz docelowe zasoby Kubernetes. Zespół rozdzielił zwykłych agentów, używanych do integracji kodu, od agentów przeznaczonych do budowania obrazów Docker, co pozwoliło lepiej dopasować uprawnienia i zasoby do rodzaju zadania.

Drugim istotnym rozwiązaniem było użycie Cloudflare Tunnel zamiast bezpośredniego wystawienia klastra do internetu. Ruch zewnętrzny trafiał najpierw do Cloudflare, następnie do warstwy ingress i NGINX, a dopiero później do wybranej usługi w klastrze. Takie podejście uprościło konfigurację dostępu z zewnątrz oraz ograniczyło konieczność publicznej ekspozycji serwera klastra.

7.5. Integracja z systemem ERP enova365

Platforma Wellhicle nie wystawia faktur ani nie prowadzi własnej ewidencji księgowej – formalne rozliczenie rezerwacji i subskrypcji przekazano dojrzałemu systemowi klasy ERP, czyli enova365 (geneza i decyzja: rozdział *3.6*). Niniejszy podrozdział opisuje rozwiązanie techniczne, które ten styk realizuje.

Stykiem obu systemów jest dedykowany dodatek do enova365 – osobny komponent, utrzymywany w odrębnym repozytorium i rozwijany niezależnie od serwera aplikacyjnego. Dodatek sięga po dane platformy poprzez jej publiczne API i odsyła do niej wyniki, co umożliwia dwukierunkową wymianę informacji. Różni to opisywaną integrację od przypadków przedstawionych w podrozdziale *7.1.11*, gdzie serwer aplikacyjny pełnił rolę strony zależnej lub dostawcy interfejsu; tutaj konsumentem API platformy jest komponent działający wewnątrz systemu ERP.

Dodatek nie został opracowany od podstaw. Powstał on w oparciu o uniwersalny wzorzec architektury dodatków integracyjnych, wypracowany przez firmę GIAX na potrzeby integrowania systemu enova365 z rozwiązaniami klasy *OMS* (*Order Management System* – systemami do zarządzania zamówieniami i ich obsługą). Punktem wyjścia był istniejący dodatek należący do tej rodziny, natomiast autorskim wkładem projektu było dostosowanie tego wzorca do modelu rozliczeniowego platformy oraz zastąpienie ręcznie napisanego klienta API komponentem

wykorzystującym typowany pakiet kliencki generowany na podstawie kontraktu (opisany w podpunkcie *7.1.11.3*).

7.5.1. System enova365 i model rozszerzeń

Enova365 to modułowy system klasy ERP firmy Soneta, zbudowany w technologii .NET (C#), z danymi przechowywanymi w relacyjnej bazie Microsoft SQL Server. Obejmuje obszary takie jak handel, finanse i księgowość czy kadry i płace, a dostępny zakres funkcjonalności zależy od wykupionej licencji [141].

Architektura systemu opiera się na modelu trójwarstwowym, obejmującym: warstwę danych, logikę biznesową oraz prezentacje (aplikacja okienkowa, przeglądarkowa i mobilna). Dostęp do danych realizowany jest poprzez autorskie mapowanie obiektowo-relacyjne. Proces modyfikacji odbywa się w ramach jednowątkowych sesji transakcyjnych, zakończonych zapisem zmian – ten mechanizm współpracuje również z kodem dodatku. Część klas logiki biznesowej generowana jest automatycznie podczas kompilacji na podstawie deklaratywnego opisu struktur danych zawartego w pliku *business.xml*, co umożliwia definiowanie nowych tabel oraz uzupełnianie ich o niestandardowy kod [142] [143] [144].

Z punktu widzenia integracji kluczowy jest model rozszerzeń, pozwalający dodawać funkcje bez modyfikowania kodu producenta. Rozszerzenie ma postać biblioteki .NET ładowanej przy starcie systemu i udostępnia m.in. *Worker* (akcja operatora w menu czynności), *Extender* (rozszerzanie klas biznesowych o pola i zachowania), własne węzły drzewa konfiguracji oraz rejestrację modułu i struktur w plikach *business.xml* i *config.xml*. Te mechanizmy stanowią fundament opisywanego dodatku [145] [146].

7.5.2. Założenia i zakres integracji

Punktem wyjścia integracji była decyzja o nieimplementowaniu w platformie własnej warstwy fakturowania i ewidencji księgowej. Budowa takiej warstwy oznaczałaby powielanie dojrzałej funkcjonalności systemu ERP oraz konieczność samodzielnego utrzymywania zgodności z przepisami podatkowymi i rachunkowymi. Zamiast tego przyjęto, że platforma odpowiada za rejestrowanie zdarzeń gospodarczych – rezerwacji po stronie wypożyczalni oraz subskrypcji platformy – a ich formalne rozliczenie powierza systemowi enova365.

Zakres wymiany danych wynika wprost z tego podziału obowiązków i obejmuje dwa kierunki. W stronę systemu ERP przekazywane są dane niezbędne do wystawienia i rozliczenia dokumentów: oferty oraz dodatki, które stają się pozycjami sprzedaży, plany subskrypcyjne oraz rachunki gromadzące zdarzenia do rozliczenia. W stronę platformy wracają wyniki tego rozliczenia – wystawione faktury w postaci dokumentów oraz aktualizacja statusów rozliczeń. Dzięki temu dokument księgowy powstaje w systemie wyspecjalizowanym do tego zadania, a platforma otrzymuje potwierdzenie rozliczenia i kopię dokumentu, nie prowadząc własnej księgowości.

7.5.3. Architektura dodatku

Dodatek można rozpatrywać w dwóch warstwach: tego, co wynika z firmowego wzorca i powtarza się w kolejnych integracjach, oraz tego, co stanowi autorski wkład tej konkretnej realizacji. Rozróżnienie to pozwala wskazać, które rozwiązania zaczerpnięto z dorobku zespołu, a które opracowano w ramach projektu.

Z firmowego wzorca pochodzi konwencja budowy dodatku: własny moduł z drzewem konfiguracji, rozszerzanie encji enova365 przez deklaracje w pliku *business.xml*, udostępnianie operacji jako akcji operatora oraz oddzielenie odwzorowania danych do osobnych komponentów mapujących. Ze wzorca wywodzi się też szkielet synchronizacji, w dodatku dla platformy Wellhicle uproszczony do węższego zakresu wymienianych danych. Wspólne biblioteki narzędziowe rodziny wykorzystano w zakresie potrzebnym tej integracji; ich szczegóły wykraczają poza temat pracy.

Autorskim wkładem tej integracji jest przede wszystkim sposób komunikacji z platformą. Zamiast ręcznie pisanego klienta sieciowego, samodzielnie obsługującego żądania i odczyt odpowiedzi, dodatek korzysta z jednego konektora konsumującego typowany pakiet kliencki generowany z kontraktu API platformy (opisany w podpunkcie *7.1.11.3*). Konektor ukrywa za wąskim interfejsem operacje takie jak logowanie, pobranie rachunków, ofert, dodatków i planów, aktualizację statusów oraz przesłanie dokumentu faktury, a uwierzytelnianie realizuje tokenem uzyskiwanym przy nawiązaniu połączenia. Oparcie o pakiet generowany z kontraktu ogranicza ryzyko rozejścia się warstwy komunikacji z rzeczywistym API platformy, a obsługę stronicowania odpowiedzi zrealizowano na modelach pochodzących z tego pakietu. Autorskim elementem jest również własny węzeł konfiguracji dodatku oraz dopasowanie zakresu synchronizacji do pojęć modelu rozliczeniowego platformy.

Powierzchnię tej komunikacji przedstawia *Listing 83* – wąski interfejs konektora, za którym ukryto operacje wykonywane na platformie. Typy występujące w jego sygnaturach pochodzą z wygenerowanego pakietu klienckiego, nie są więc pisane ręcznie.

Listing 83 Interfejs konektora platformy Wellhicle – operacje wymiany danych. Źródło: opracowanie własne.

```
public interface IWellhicleConnector : IDisposable
{
    void Login();
    List<int?> GetBillingIds(BillingErpStatus status);
    BillingDto GetBilling(int id);
    void UpdateBillingsStatus(List<int> ids, BillingErpStatus? status);
    void UploadBillingDocument(int billingId, byte[] fileBytes,
        string fileName);
    List<PaymentMethodDto> GetPaymentMethods();
    PageBase<OfferListDto> GetOffersPage(int page = 1);
    PageBase<AdditionDto> GetAdditionsPage(int page = 1);
    PageBase<PlanDto> GetPlansPage(int page = 1);
}
```

Samo wywołanie operacji sprowadza się do odpytania typowanej trasy tego pakietu, co na przykładzie pobrania strony ofert pokazuje *Listing 84*. Konektor nie buduje adresu żądania ani nie serializuje danych ręcznie – korzysta z gotowej, typowanej ścieżki `_client.Api.Offers.GetAll` i sprowadza odpowiedź, możliwą do podziału na strony, do wspólnego modelu stronicowania. Wywołania są synchroniczne, ponieważ czynności enova365 wykonują się w jednym wątku, a wszystkie przechodzą przez jednolitą obsługę błędów tłumaczącą wyjątki na czytelny komunikat dla operatora.

Listing 84 Pobranie strony ofert przez typowaną trasę wygenerowanego klienta – bez ręcznego budowania adresu i serializacji, z ujednoliconą obsługą błędów. Źródło: opracowanie własne.

```
public PageBase<OfferListDto> GetOffersPage(int page = 1)
{
    return Execute(() => _client.Api.Offers.GetAll.GetAsync(
            requestConfiguration =>
            {
                requestConfiguration.QueryParameters.Page = page;
                requestConfiguration.QueryParameters.PageSize = pageSize;
            }, cancellationToken: default
        ).GetAwaiter().GetResult()?.ToPage()
        ?? new PageBase<OfferListDto>(),
```

```
            $"Pobranie strony {page} ofert z Wellhicle nie powiodlo sie"
        );
    }
```

7.5.4. Model danych i mapowanie na encje enova365

Aby dane pochodzące z platformy mogły funkcjonować w systemie ERP, dodatek rozszerza wybrane encje enova365 o informacje potrzebne do powiązania ich z obiektami platformy. Rozszerzenia te zadeklarowano w pliku *business.xml* jako tabele dowiązane do encji systemu, tak że każdy rekord rozszerzenia jest trwale związany ze swoim obiektem-gospodarzem. W ten sposób towar, dokument handlowy oraz pozycja dokumentu zyskują swoje odpowiedniki przechowujące dane integracji, uzupełnione o struktury konfiguracyjne, definicję dokumentu oraz sposób zapłaty i operacyjny zapis płatności.

Rozszerzenia opisuje się deklaratywnie w pliku *business.xml*, a klasy wiersza i tabeli powstają z tego opisu w czasie kompilacji – w ramach budowania dodatku pakietem Soneta.Sdk. Dodatek nie ingeruje więc w schemat bazy danych ani w kod producenta: deklaruje własną tabelę-rozszerzenie, a wygenerowane dla niej klasy uzupełnia logiką. Przykład takiej deklaracji, dla rozszerzenia encji towaru, przedstawia *Listing 85*.

Listing 85 Deklaracja rozszerzenia encji towaru w pliku business.xml – tabela TowarWellhicle dowiązana do encji Towar. Źródło: opracowanie własne.

```
<table name="TowarWellhicle" tablename="TowaryWell">
    <col name="Host" type="Towar" keyprimary="true"
        keyunique="true" relguided="inner"
        relname="TowarWellhicle" delete="cascade"
        readonly="true" relright="true"
        required="true"
    />
    <col name="Aktywny" type="boolean" modifier="public virtual"/>
    <col name="Type" type="WellhicleModels.BillingItemType"/>
    <col name="ProductId" type="int"/>
    <col name="SourceContent" type="text" modifier="public virtual"/>
    <key name="WgProductId">
        <keycol name="Type"/>
        <keycol name="ProductId"/>
    </key>
</table>
```

Rozszerzenie jest powiązane z encją-gospodarzem przez kolumnę `Host`, połączoną z encją `Towar` relacją prowadzoną typu wewnętrznego: rekord rozszerzenia dzieli klucz główny gospodarza i powstaje wraz z nim, a kaskadowe usuwanie sprawia, że znika razem z nim. Samą rejestrację rozszerzenia w systemie zapewnia atrybut pochodzący z bibliotek wzorca, tak że każdej encji towaru towarzyszy jej rekord integracji, dostępny w kodzie przez akcesor wywoływany na obiekcie-gospodarzu (`towar.Wellhicle()`). W rezultacie dane integracji stają się częścią modelu obiektowego enova365 i podlegają jego regułom zapisu, mimo że nie naruszają oryginalnych struktur systemu.

Najważniejszą rolą tych rozszerzeń jest utrzymanie powiązania między obiektem w systemie ERP a jego źródłem w platformie. Służą temu pola korelacyjne przechowujące identyfikatory pochodzące z platformy – identyfikator produktu, rachunku, najemcy czy płatności – oraz status rozliczenia. Obok nich każdy rozszerzony rekord przechowuje surową treść danych źródłowych w postaci tekstowej; zapis ten pełni rolę pomocniczą, ułatwiając diagnostykę i prześledzenie, na podstawie jakich danych powstał dany obiekt. Pola korelacyjne pozwalają też przy kolejnym uruchomieniu synchronizacji rozpoznać, czy dany element platformy ma już swój odpowiednik w enova365, i zaktualizować go zamiast tworzyć duplikat.

Odwzorowanie danych platformy na obiekty enova365 realizują komponenty mapujące, które przepisują wartości z obiektów transferowych pakietu klienckiego na właściwości encji systemu. Szczególnym przypadkiem jest uzgadnianie towarów: wyodrębniony komponent dopasowujący odnajduje istniejący towar odpowiadający ofercie lub dodatkowi platformy albo zakłada nowy, posługując się polami korelacyjnymi. Odpowiedzi API platformy, które mogą obejmować wiele stron wyników, są odczytywane przy użyciu wspólnej obsługi stronicowania, tak że mapowanie operuje na kompletnym zbiorze danych niezależnie od podziału na strony. Poglądowe zestawienie głównych powiązań między pojęciami platformy a encjami enova365 przedstawia *Tabela 61*.

Tabela 61 Powiązanie pojęć platformy Wellhicle z encjami systemu enova365. Źródło: opracowanie własne.

Pojęcie platformy	**Odwzorowanie w enova365**
Oferta, Dodatek, Plan subskrypcyjny	Towar
Rachunek	Dokument handlowy
Pozycja rachunku	Pozycja dokumentu handlowego
Faktura (wynik)	Dokument handlowy (faktura)

7.5.5. Procesy synchronizacji i wymiana dokumentów

Wymianę danych organizuje wspólny szkielet synchronizacji. Każdy rodzaj synchronizowanych danych ma swoją pozycję synchronizacji, wybieraną na podstawie określonego typu, a wszystkie pozycje dzielą jeden mechanizm raportowania przebiegu oraz obsługi błędów. Pozycja synchronizacji udostępnia dwie operacje odpowiadające dwóm kierunkom wymiany – pobranie danych z platformy do systemu ERP oraz przekazanie wyników z systemu ERP do platformy – przy czym konkretna pozycja implementuje tylko ten kierunek, który ma sens dla danego rodzaju danych. W trakcie pracy mechanizm raportuje postęp i komunikaty do operatora, a odczyt danych z platformy odbywa się stronami, z zabezpieczeniem przed nadmierną liczbą żądań.

W kierunku do systemu ERP dodatek pobiera z platformy oferty, dodatki i plany subskrypcyjne, tworząc z nich lub aktualizując odpowiadające im towary, oraz rachunki, na podstawie których powstają dokumenty handlowe wraz z pozycjami i płatnościami. Przebieg jest w każdym przypadku podobny: dla kolejnych elementów pobranej strony wyników dodatek odnajduje istniejący obiekt enova365 albo zakłada nowy, po czym przepisuje na niego dane źródłowe i zapisuje powiązanie. Pomijane są przy tym dane niekompletne, na przykład pozycje bez identyfikatora lub bez nazwy, co odnotowywane jest w wyniku synchronizacji.

W kierunku do platformy odsyłane są rezultaty rozliczenia. Dla dokumentu reprezentującego rozliczone zamówienie dodatek tworzy w systemie enova365 fakturę, generuje jej wydruk w postaci pliku przy użyciu usługi raportowania pochodzącej z bibliotek wzorca, a następnie przesyła tak przygotowany dokument do platformy, wiążąc go z właściwym rachunkiem. Odrębna pozycja synchronizacji odpowiada za aktualizację statusów rozliczeń – przekazuje platformie informację, że wskazane rachunki zostały zafakturowane. Dzięki temu po stronie platformy widoczne jest zarówno potwierdzenie rozliczenia, jak i sam dokument faktury.

Operator uruchamia wymianę z poziomu interfejsu enova365, korzystając z akcji udostępnianych przez dodatek – osobnych dla pobrania poszczególnych rodzajów danych oraz dla wystawienia i odesłania faktury. Pozwala to kontrolować moment i zakres synchronizacji oraz obserwować jej przebieg.

7.5.6. Konfiguracja, uruchomienie i automatyzacja

Uruchomienie integracji wymaga jednorazowej konfiguracji połączenia po stronie enova365. Dodatek dokłada do drzewa konfiguracji systemu własny węzeł, w którym wskazuje się adres API platformy, źródło zgłoszenia oraz poświadczenia używane do logowania, a w osobnej części – ustawienia związane z wystawianiem faktur, w tym wzorzec wydruku dokumentu. Definicję tego węzła zapisano deklaratywnie – podobnie jak model danych – w osobnym pliku konfiguracyjnym dodatku, z którego podczas kompilacji powstają typowane klasy dostępu do ustawień. Poprawność wprowadzonych danych można zweryfikować wbudowanym sprawdzeniem połączenia, które nawiązuje próbną komunikację z platformą i zwraca czytelny komunikat o jej wyniku. Tak przygotowana konfiguracja jest następnie wykorzystywana przez konektor przy każdym uruchomieniu synchronizacji.

W trybie obsługiwanym ręcznie integracja sprowadza się do wywoływania akcji udostępnianych przez dodatek w interfejsie systemu. Pobranie danych oraz wystawienie i odesłanie faktury są osobnymi operacjami, tak że obsługa zachowuje kontrolę nad tym, kiedy i jakie dane są synchronizowane, a przebieg każdej operacji jest raportowany. Taki tryb pracy ułatwia wdrażanie i diagnostykę, gdy pożądana jest pełna kontrola nad momentem wykonania każdej czynności.

Operacje dodatku udostępniono jako standardowe czynności systemu enova365. Oprócz wywołania ręcznego mogą one być uruchamiane cyklicznie przez systemowy harmonogram zadań enova365, bez logowania operatora i tę właściwość wykorzystano do pracy bezobsługowej. W środowisku demonstracyjnym przygotowanym na potrzeby pracy wystawianie i odsyłanie faktur skonfigurowano jako zadanie cykliczne harmonogramu, tak że dla rozliczonego zamówienia faktura powstaje i wraca do platformy wraz ze statusem rozliczenia bez udziału operatora. Automatyzacja oparła się na wbudowanym mechanizmie platformy i nie wymagała dodatkowego kodu po stronie dodatku; rola człowieka ogranicza się do wcześniejszej konfiguracji.

7.5.7. Stan rozwiązania i kierunki rozwoju

Opisana integracja jest rozwiązaniem działającym i spełnia swoją podstawową potrzebę: na podstawie rozliczenia powstałego w platformie wystawia w systemie enova365 fakturę i odsyła ten dokument wraz ze statusem z powrotem do platformy. W tym zakresie dodatek realizuje pełny, dwukierunkowy obieg danych opisany w poprzednich podrozdziałach. Należy go jednak traktować jako rozwiązanie zrealizowane w ramach pracy inżynierskiej i potwierdzające założony scenariusz, a nie jako produkt wdrożony u klienta.

Wyznacza to granicę obecnego zakresu. Wdrożenie integracji u konkretnego klienta wymagałoby przede wszystkim konfiguracji dostosowanej do jego środowiska: parametrów połączenia, definicji dokumentów, sposobów zapłaty i wzorców wydruku faktur. W zależności od potrzeb klienta konieczne mogłoby okazać się dopracowanie rozwiązania lub rozszerzenie zakresu wymienianych danych poza obecnie obsługiwane oferty, dodatki, plany subskrypcyjne, rachunki i faktury. Osobnym zagadnieniem jest obieg dokumentów po stronie systemu ERP: w jednym wdrożeniu właściwe może być utworzenie dedykowanego obiegu dla dokumentów pochodzących z platformy, w innym – wpięcie się w obieg już funkcjonujący u klienta. Tego rodzaju decyzje zależą od specyfiki konkretnego przedsiębiorstwa i wykraczają poza zakres niniejszej pracy.

Przyjętej architekturze sprzyja oparcie na standardowych mechanizmach rozszerzeń enova365: rozszerzenie zakresu danych sprowadza się zwykle do dołożenia kolejnych elementów synchronizacji i mapowań, a dostosowanie do środowiska klienta – do samej konfiguracji, bez ingerencji w trzon rozwiązania. Z tego samego oparcia korzysta tryb bezobsługowy: cykliczne wystawianie i odsyłanie faktur realizuje wbudowany harmonogram zadań enova365, bez dodatkowego kodu po stronie dodatku.

W zrealizowanym zakresie integracja pozwala platformie skupić się na rejestrowaniu zdarzeń gospodarczych i powierzyć ich formalne rozliczenie wyspecjalizowanemu systemowi ERP.

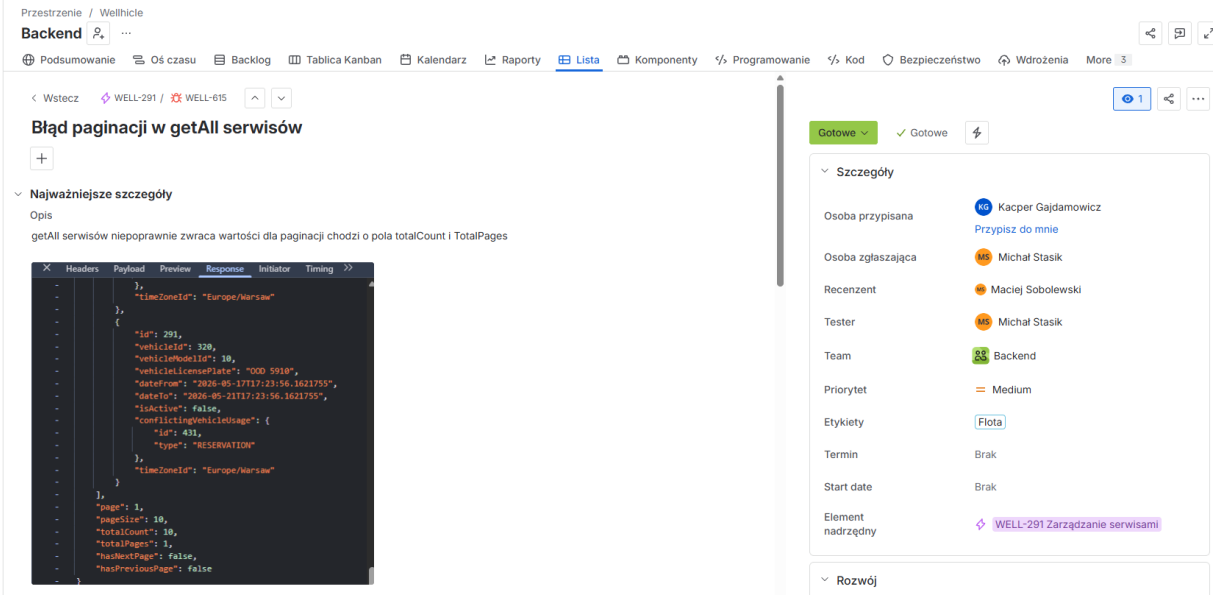
8. Testy

Testowanie stanowi istotny etap wytwarzania oprogramowania, pozwalając na wykrycie błędów, weryfikację spełnienia wymagań systemu oraz ocenę stabilności rozwiązania co bezpośrednio przekłada się na jego jakość. W poniższym rozdziale zostały opisane testy jakie przeprowadzono w ramach weryfikacji poprawnego działania systemu Wellhicle wsparte przykładami implementacji oraz raportami przeprowadzonych testów.

Podczas procesu testowania zespół testerów wspomagał się scenariuszami przypadków użycia powstałymi podczas fazy analizy. Prezentowały one wszystkie najważniejsze funkcjonalności oraz przypadki alternatywne, które użytkownik mógł wywołać.

Do testów zespół wykorzystywał trzy środowiska uruchomieniowe. Środowisko produkcyjne z najnowszą stabilną formą systemu. Środowisko deweloperskie zawierające najnowszą przedprodukcyjną wersję aplikacji, zarówno części serwerowej jak i interfejsu użytkownika. Dodatkowo każdy z członków zespołu mógł skorzystać z własnego środowiska uruchomionego lokalnie pobierając kod bezpośrednio z repozytorium GitHub.

W sytuacji napotkania błędu, tester rejestrował jego wystąpienie poprzez utworzenie zgłoszenia typu „Bug” w systemie Jira, łącząc nowopowstałe zgłoszenie do zgłoszenia nadrzędnego reprezentującego daną funkcjonalność, z którą związany jest błąd. Tester podawał przy tym wszelkie niezbędne informacje opisujące warunki powstania błędu, uzupełniane były również metadane zgłoszenia m.in.: osoba przypisana, tester, zespół, na którego kolejkę zostanie dodane zgłoszenie, priorytet, oraz element nadrzędny (zgłoszenie nadrzędne). *Rysunek 157* przedstawia zrzut ekranu z systemu Jira z widokiem zarejestrowanego błędu w oprogramowaniu.



Rysunek 157 Przykładowy widok zgłoszonego błędu. Źródło: opracowanie własne.

Więcej przydatnych informacji dotyczących modelowania systemów informatycznych, w tym zagadnień związanych z testowaniem oprogramowania, doborem odpowiednich narzędzi wspierających ten proces, można znaleźć w [147].

8.1. Testy manualne i integracyjne

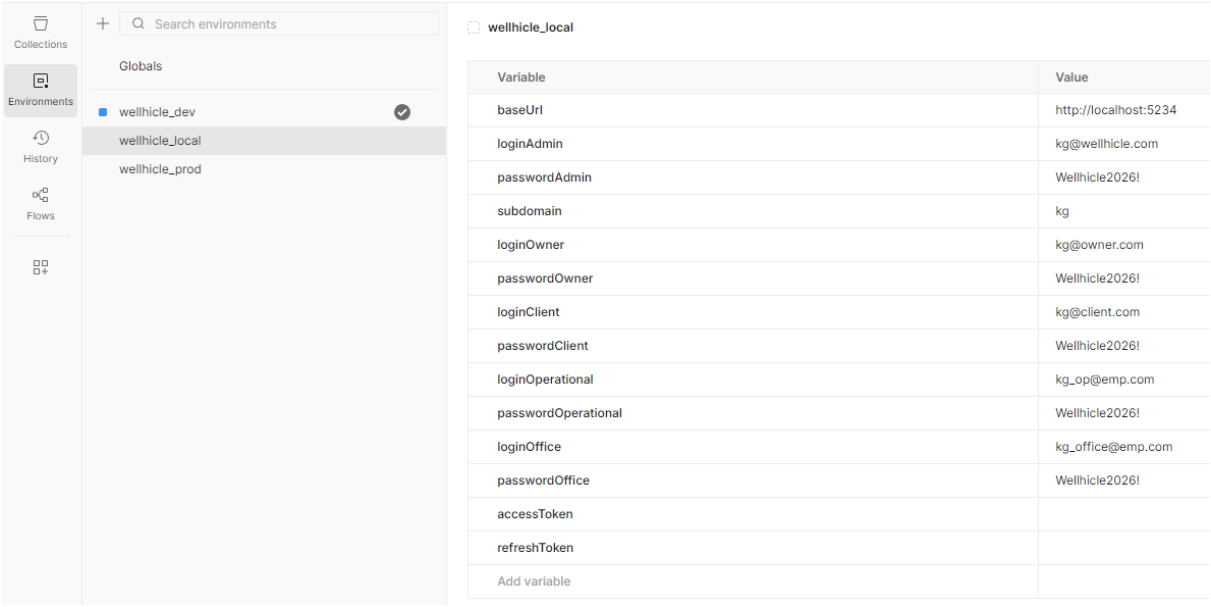
W ramach weryfikacji poprawności działania systemu Wellhicle przeprowadzono testy manualne oraz integracyjne, obejmujące zarówno warstwę serwerową, jak i kliencką aplikacji. Celem

opisanych w niniejszym rozdziale działań było potwierdzenie, że poszczególne moduły systemu działają poprawnie nie tylko indywidualnie, lecz również w ramach komunikacji między warstwami aplikacji.

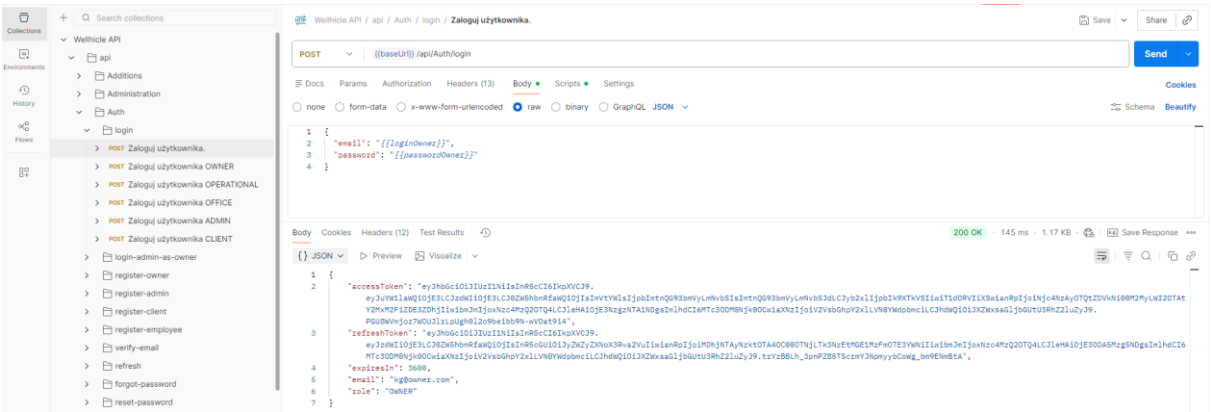
8.1.1. Testy integracyjne

Testy API przeprowadzono głównie przy użyciu dwóch narzędzi – Postman i Scalar, opisanych kolejno w rozdziałach 4.7.1 oraz 4.7.4. W procesie testowania zastosowano podejście czarnej skrzynki zgodnie z którą testerzy nie znali struktury kodu źródłowego testowanego modułu. Weryfikacji poddawano wyłącznie zewnętrzne zachowanie systemu – poprawność zwracanych danych a także kodów odpowiedzi HTTP oraz obsługę przypadków błędnych.

Zapytania pogrupowano w kolekcje co zapewniło przejrzystą organizację zestawu testowanych punktów końcowych. W celu eliminacji konieczności ręcznej modyfikacji parametrów przy przełączaniu między środowiskiem lokalnym a deweloperskim, skonfigurowano zmienne środowiskowe przechowujące m.in. adresy bazowe, loginy poszczególnych użytkowników oraz ich hasła. *Rysunek 158* przedstawia widok konfiguracji zmiennych środowiskowych w Postmanie natomiast na *Rysunek 159* zademonstrowano ich wykorzystanie w ciele żądania.



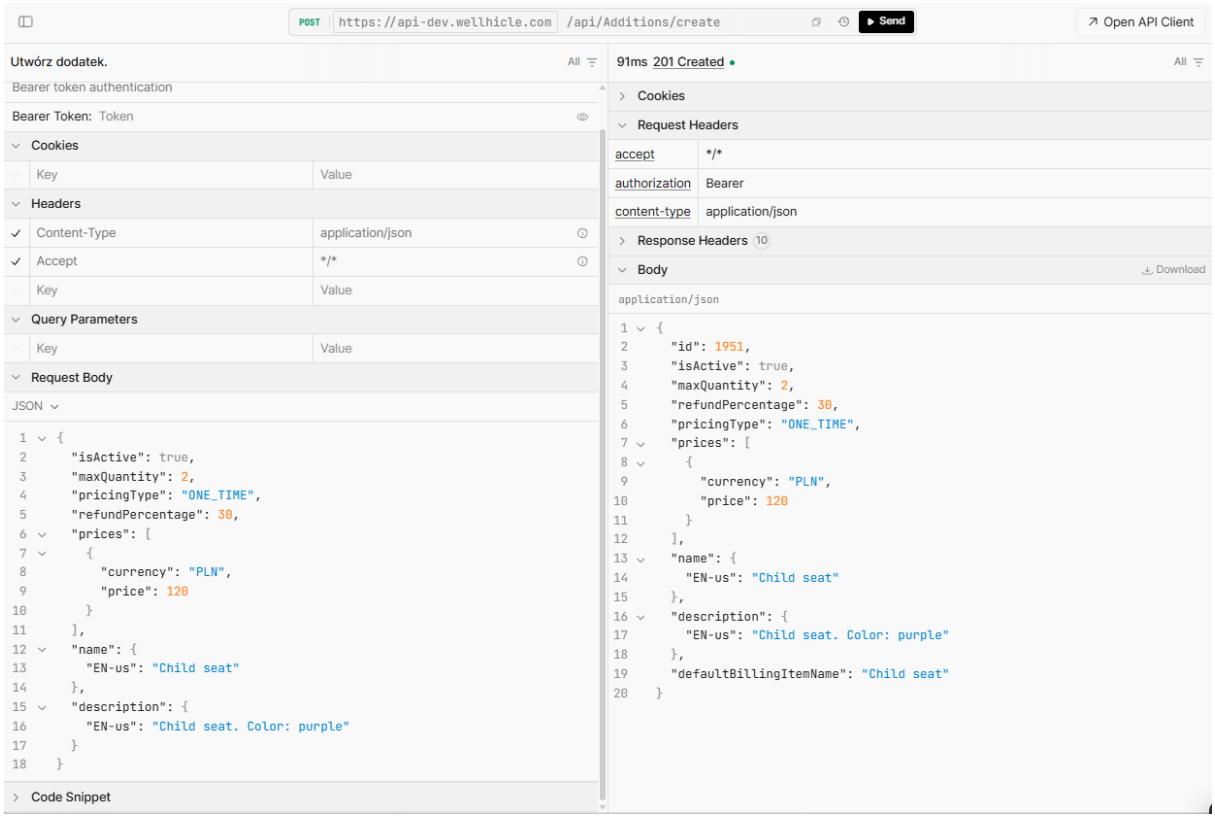
Rysunek 158 Konfiguracja zestawu zmiennych środowiskowych w narzędziu Postman. Źródło: opracowanie własne.



Rysunek 159 Wykorzystanie zmiennych środowiskowych w ciele żądania. Źródło: opracowanie własne.

Dla zapytań wymagających uwierzytelnienia zastosowano mechanizm post-response scripts, który umożliwił pobranie i zapisanie tokenu uwierzytelniającego do zmiennej. Wyeliminowało to ryzyko wykonania testu z wygasłym tokenem i ograniczyło liczbę manualnych interakcji podczas sesji testowych.

W celu przeprowadzania prostych testów wykorzystywano Scalara. Ze względu na wbudowanego klienta HTTP narzędzie umożliwiało natychmiastowe wysłanie zapytania bez konieczności konfigurowania dedykowanej kolekcji, co sprawdzało się szczególnie podczas bieżących prac implementacyjnych, gdy zachodziła potrzeba doraźnego sprawdzenia działania danego punktu końcowego mając jednocześnie otwartą dokumentację. Przykład użycia Scalara przedstawiono na *Rysunek 160*.



Rysunek 160 Przykładowe użycie Scalara w celu przetestowania API. Źródło: opracowanie własne.

8.1.2. Testy manualne

Testy manualne przeprowadzono dla aplikacji przeglądarkowych oraz aplikacji mobilnej. Celem weryfikacji było odwzorowanie rzeczywistego zachowania użytkownika w trakcie korzystania z systemu, ze szczególnym uwzględnieniem bardziej złożonych przypadków użycia. Do przetestowania danego obszaru dobierano członka zespołu, który nie był zaangażowany w proces tworzenia danej funkcjonalności co oznaczało, że nie posiadał wiedzy na temat budowy kodu źródłowego testowanego modułu. Taki dobór testerów umożliwił rzetelne sprawdzenie poprawnego działania systemu.

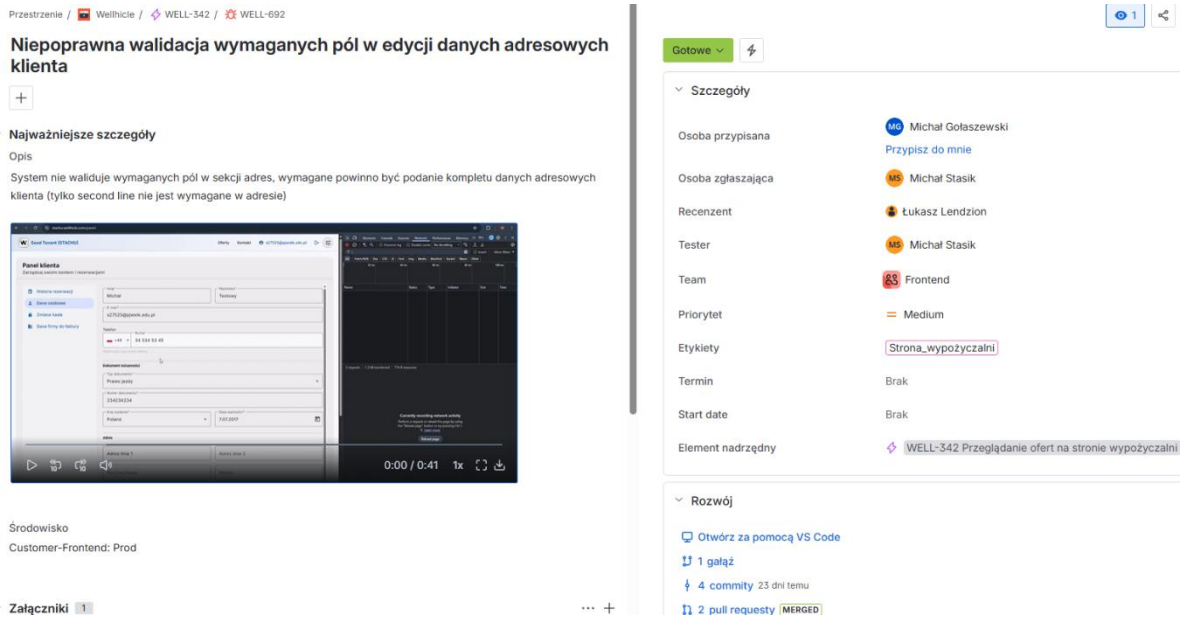
Do procesu dokumentowania przeprowadzonych testów wykorzystano raport, którego struktura powstała na podstawie dokumentu wytworzonego podczas fazy analizy. Dokument zawierał spis funkcjonalności przypisanych do poszczególnych modułów, co umożliwiło systematyczne pokrycie wszystkich wymagań funkcjonalnych. Dla każdej funkcjonalności tester odnotowywał jej wynik w kolumnie „Sukces” – prawda w przypadku jej poprawnego działania lub fałsz w przypadku znalezienia

błędów. *Tabela 62* przedstawia fragment raportu z widocznymi wynikami testów modułu strony publicznej wypożyczalni.

Tabela 62 Fragment raportu testów manualnych strony publicznej wypożyczalni. Źródło: opracowanie własne.

Funkcjonalność	**Aktor**	**Znalezione błędy**	**Link do Jira**	**Status rozwiązania błędu**	**Sukces**	**Zaplanowane w analizie**
Prezentacja oferty pojazdów	Gość				PRAWDA	TAK
Filtrowanie i wyszukiwanie ofert	Gość / Klient				PRAWDA	TAK
Szczegóły oferty	Gość / Klient				PRAWDA	TAK
Konfiguracja wstępna rezerwacji	Gość / Klient	Brak zmiany waluty dodatków	https://wellhicle.atlassian.net/browse/WELL-689	Rozwiązano	FAŁSZ	TAK
Rejestracja i logowanie klienta (frontend)	Gość / Klient	Brak wymogu akceptacji regualminu	https://wellhicle.atlassian.net/browse/WELL-691	Rozwiązano	FAŁSZ	TAK
Finalizacja rezerwacji online	Klient wypożyczalni	Nie da się utworzyć jednodniowej rezerwacji	https://wellhicle.atlassian.net/browse/WELL-690	Rozwiązano	FAŁSZ	TAK

W przypadku wykrycia błędu tester rejestrował jego wystąpienie w systemie Jira, tworząc zgłoszenie typu „Bug" z odwołaniem do zgłoszenia nadrzędnego reprezentującego daną funkcjonalność, zgodnie z procesem opisanym we wstępie rozdziału *8.* Odnośnik do zgłoszenia był następnie odnotowywany bezpośrednio w raporcie w kolumnie „Link do Jira". Uzupełniane były również kolumny „Znalezione błędy”, zawierającą krótki opis błędu oraz w późniejszym etapie „Status rozwiązania błędu”, która zawierała informację czy zgłoszony błąd został rozwiązany czy odrzucony np. z powodu nieprawidłowej identyfikacji błędu jako poprawnego działania systemu. *Rysunek 161* przedstawia przykładowe zgłoszenie błędu zarejestrowane w trakcie testów manualnych.



Rysunek 161 Przykład błędu wykrytego podczas testów manualnych. Źródło: opracowanie własne.

8.2. Testy automatyczne

Testy automatyczne zrealizowano z wykorzystaniem narzędzia Playwright opisane w rozdziale Playwright. Celem testów była weryfikacja poprawnego działania podstawowych funkcjonalności systemu z perspektywy interfejsu graficznego – sprawdzenie czy użytkownik jest w stanie za jego pomocą wykonać kluczowe operacje. Testy napisano w języku TypeScript, stosując wzorzec projektowy Page Object Model (POM), który zakłada wyodrębnienie logiki interakcji z poszczególnymi widokami aplikacji w dedykowanych klasach [148]. Podejście to umożliwiło oddzielenie kodu odpowiedzialnego za nawigację i manipulację elementami interfejsu od kodu scenariuszy testowych, co przełożyło się na większą czytelność i łatwość utrzymania zestawu testów. Przykładowy fragment klasy `AdditionPage` implementująca wzorzec POM, zawierająca lokatory elementów interfejsu oraz metody pomocnicze, przedstawiona została na Listing 86.

Listing 86 Fragment klasy `AdditionPage` implementująca wzorzec POM. Źródło: opracowanie własne.

```
import { Locator, Page, expect } from "@playwright/test";

export class AdditionPage {
    readonly page: Page;
    readonly additionLink: Locator;
    readonly addButton: Locator;
    readonly saveButton: Locator;
    …

    constructor(page: Page) {
       this.page = page;
       this.additionLink = page.getByTestId("nav-additions");
       this.addButton = page.getByTestId("addition-add-button");
       this.saveButton = page.getByTestId("addition-save-button");
       this.snackBar = page.locator("simple-snack-bar");
       …
    }

    async navigateToAdditions() {
       await this.additionLink.click();
    }

    async openAddDialog() {
       await this.addButton.click({ force: true });
    }
    async openDetails(name: string) {
       await this.page.getByRole("cell", { name: name, exact: true
}).click();
    }
    async fillTranslationFields(lang: string, name: string, description:
string) {
       const tab = this.page.getByTestId(`tab-${lang}`);
       const nameInput = this.page.getByTestId(`input-name-${lang}`);
       const descInput = this.page.getByTestId(`input-description-
${lang}`);
       await tab.click();
       await expect(nameInput).toBeVisible();
       await nameInput.fill(name);
       await descInput.fill(description);
    }
```

Ważnym aspektem konfiguracji zestawu testowego było zachowanie określonej kolejności wykonywania projektów testowych, która wynikała z zależności między modułami systemu. Konfiguracja Playwrighta zakładała podział testów na projekty odpowiadające poszczególnym modułom, przy czym każdy projekt definiował jawne zależności od projektów poprzedzających go. Przykładowo, testy modułu zarządzania pracownikami mogły zostać uruchomione dopiero po pomyślnym wykonaniu testów modułu zarządzania oddziałami, ponieważ dane wytworzone w trakcie tych ostatnich były wymagane do prawidłowego przeprowadzenia scenariuszy dotyczących pracowników. Fragment konfiguracji zależności pomiędzy projektami testowymi przedstawiono na *Listing 87*.

Listing 87 Fragment pliku konfiguracyjnego Playwright z uwzględnieniem zależności pomiędzy zestawami testów. Źródło: opracowanie własne.

```
projects: [
    {
       name: "employees-happyPath",
       testMatch: /Employee\.spec\.ts/,
       grep: /@critical/,
       use: { ...devices["Desktop Chrome"], storageState: "auth.json" },
       dependencies: ["depts-happyPath"],
    },
    {
       name: "employees-validation",
       testMatch: /Employee\.spec\.ts/,
       grepInvert: /@critical/,
       use: { ...devices["Desktop Chrome"], storageState: "auth.json" },
       dependencies: ["depts-happyPath"],
    },

    {
       name: "addition-happyPath",
       testMatch: /Addition\.spec\.ts/,
       grep: /@critical/,
       use: { ...devices["Desktop Chrome"], storageState: "auth.json" },
       dependencies: ["setup"],
    },
    {
       name: "addition-validation",
       testMatch: /Addition\.spec\.ts/,
       grepInvert: /@critical/,
       use: { ...devices["Desktop Chrome"], storageState: "auth.json" },
       dependencies: ["addition-happyPath"],
    },
```

Analogiczna zależność zachodziła między pozostałymi modułami systemu. Stan współdzielony między projektami był przekazywany za pośrednictwem pliku JSON, do którego poszczególne scenariusze zapisywały identyfikatory i dane obiektów tworzonych w trakcie wykonania.

W ramach każdego modułu testy zostały podzielone na dwie grupy: scenariusze ścieżki głównej (ang. *Happy path*), oznaczone tagiem „critical", weryfikujące podstawowy przebieg operacji, oraz scenariusze walidacyjne sprawdzające obsługę przypadków brzegowych. Testy walidacyjne każdego modułu były uzależnione od uprzedniego pomyślnego wykonania jego ścieżki głównej, co gwarantowało spójność stanu danych w trakcie całego przebiegu testowego. *Listing 88* przedstawia implementację scenariusza ścieżki głównej testów modułu dodatków w panelu zarządzania wypożyczalnią, w której wykorzystano metody pomocnicze z klasy AdditionPage.

Listing 88 Implementacja ścieżki głównej testów dodatków. Źródło: opracowanie własne.

```
test.describe("Zarządzanie dodatkami (Addition)", () => {
    let additionPage: AdditionPage;
    const dataFile = "user-data.json";

    test.beforeEach(async ({ page }) => {
       additionPage = new AdditionPage(page);
       await page.goto("/");
       await additionPage.navigateToAdditions();
    });
    test("Happy Path: Utworzenie 6 dodatków @critical", async ({ page })
=> {
       const roles = ["first", "second", "third", "test1", "test2",
"test3"] as const;
       const additionsInfo: any = {};
```

```
        for (const role of roles) {
            const additionData = {
                plName: `Dodatek_${role.toUpperCase()}`,
                plDesc: `Opis dla roli ${role}`,
                enName: `Addition_${role.toUpperCase()}`,
                enDesc: `Description for ${role}`,
                price: (Math.floor(Math.random() * 50) + 10).toString(),
                maxQty: "5",
                refundPercentage: 10,
                isActive: true,
            };
            await page.waitForTimeout(300);
            await additionPage.openAddDialog();
            await additionPage.fillTranslationFields("pl-PL",
additionData.plName, additionData.plDesc);
            await additionPage.fillTranslationFields("en-US",
additionData.enName, additionData.enDesc);
            await additionPage.fillPrice("PLN", additionData.price);
            await additionPage.fillCommonFields(additionData.maxQty,
additionData.refundPercentage);
            await additionPage.saveButton.click();
            await expect(additionPage.snackBar).toContainText(/added/i);
            await additionPage.closeSnackBar();

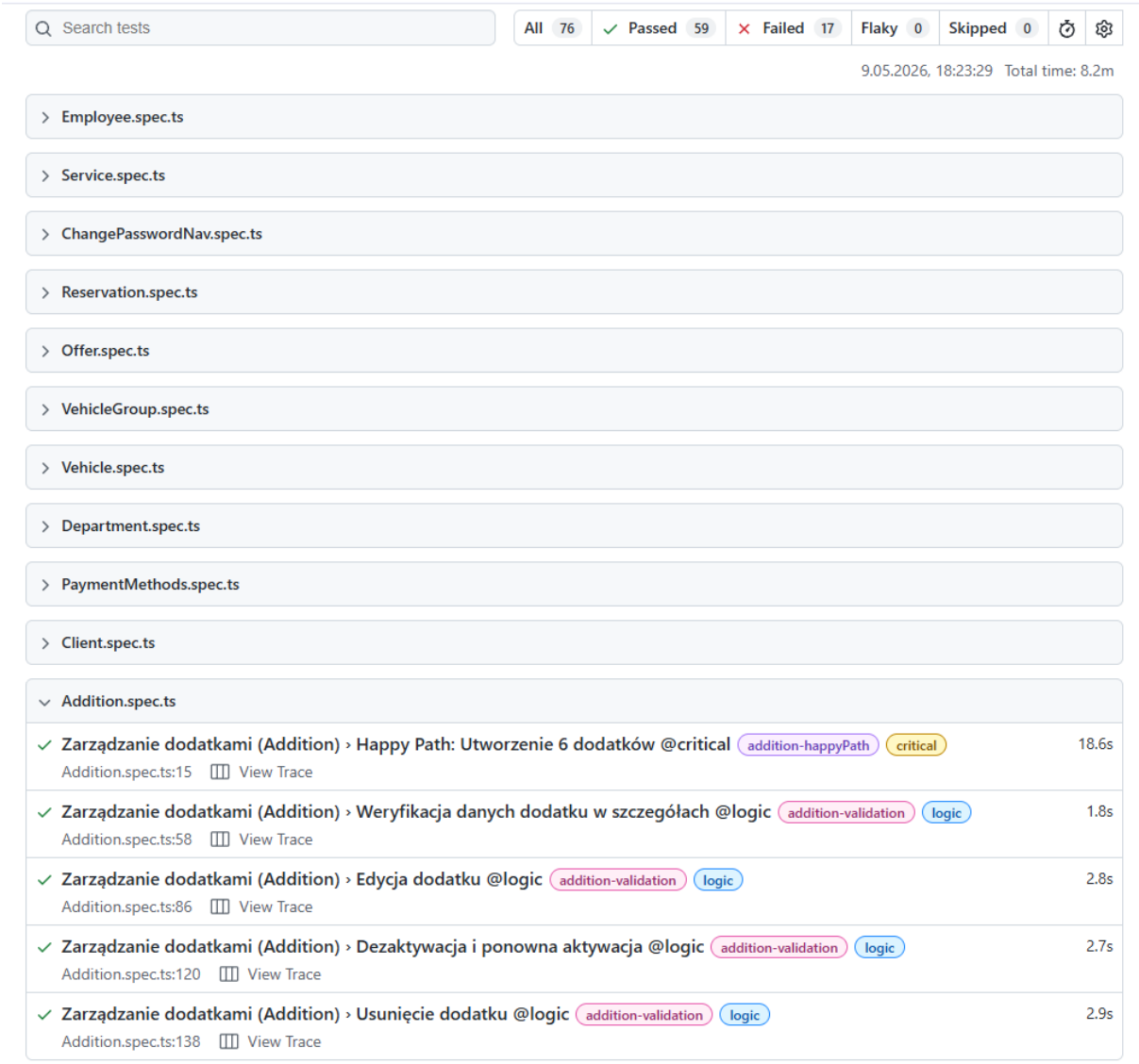
            await additionPage.openDetails(additionData.plName);
            await expect(page.getByTestId("addition-detail-
status")).toHaveText(/inactive/i);
            await additionPage.activateBtn.click();
            await additionPage.confirmAction("Activate");
            await
expect(additionPage.snackBar.last()).toContainText(/activated/i);
            await additionPage.closeSnackBar();

            additionsInfo[role] = additionData;
            await page.waitForTimeout(300);
        }

        const currentData = JSON.parse(fs.readFileSync(dataFile, "utf-
8"));
        currentData.additions = additionsInfo;
        fs.writeFileSync(dataFile, JSON.stringify(currentData, null, 2));

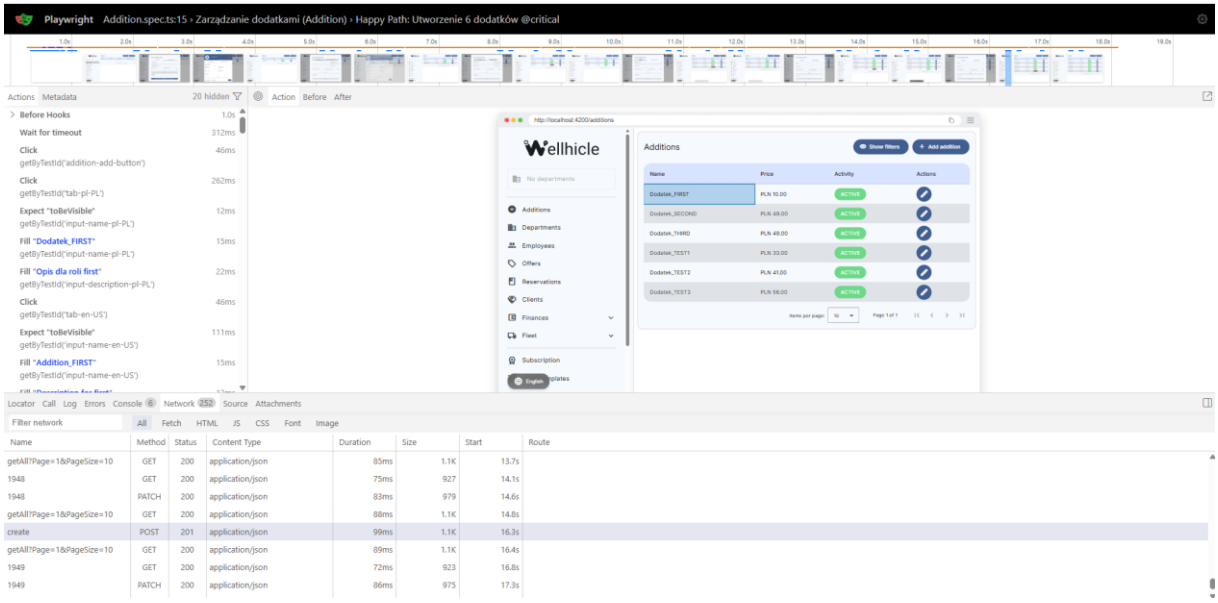
        await expect(page.getByRole("cell", { name: "Dodatek_FIRST",
exact: true })).toBeVisible();
    });
```

Dodatkową zaletą narzędzia był wbudowany raport HTML generowany po każdorazowym uruchomieniu zestawu testów. Raport zawierał pełną listę przeprowadzonych scenariuszy wraz z ich wynikami. *Rysunek 162* przedstawia raport wykonanych testów z narzędzia Playwright.



Rysunek 162 Raport wykonanych testów w narzędziu Playwright. Źródło: opracowanie własne.

Dla każdego scenariusza dostępny był zapis w formie wizualnej z podglądem wykonywanych kroków oraz stanu przeglądarki w każdym momencie wykonania, co znacznie ułatwiało identyfikację miejsca wystąpienia problemu. Raport zawierał również dane sieciowe przeglądarki, w tym zestawienie wysyłanych i odbieranych żądań HTTP poszczególnych punktów końcowych API wraz z treścią odpowiedzi, co znacznie usprawniało znalezienie przyczyny potencjalnego błędu. *Rysunek 163* prezentuje widok wybranego scenariusza z danymi sieciowymi przeglądarki.



Rysunek 163 Widok zrealizowanego testu w raporcie Playwright. Źródło: opracowanie własne.

Zestawy testów automatycznych były wartościowym źródłem weryfikacji działania systemu w toku trwających prac rozwojowych. Umożliwiły one regresyjne sprawdzenie modułów pozornie niezwiązanych z wprowadzanymi zmianami, sygnalizując nieoczekiwane efekty uboczne.

8.3. Testy jednostkowe

Testy jednostkowe stanowią podstawową formę automatycznej weryfikacji poprawności sposobu działania kodu części serwerowej systemu Wellhicle. Celem przeprowadzonych testów było zabezpieczenie reguł biznesowych warstwy domenowej oraz logiki obsługi poleceń i zapytań przed regresją, a także zapewnienie wykonywalnej dokumentacji zachowania kodu – nazwa każdej metody testowej opisuje warunek wejściowy oraz oczekiwany rezultat. Do uruchamiania testów wykorzystano bibliotekę xUnit, opisaną w podrozdziale *4.7.6.*

W procesie tworzenia testów jednostkowych wykorzystano narzędzia oparte na sztucznej inteligencji, które wspomagały generowanie kodu testowego. Na podstawie analizy kodu oraz zdefiniowanych reguł biznesowych model generował propozycje metod testowych. Każdy wygenerowany test podlegał następnie weryfikacji przez członka zespołu, który oceniał poprawność scenariusza, adekwatność asercji oraz zgodność z przyjętymi założeniami projektu. Podejście to pozwoliło na znaczące przyspieszenie pokrycia testowego przy jednoczesnym zachowaniu kontroli jakości przez człowieka.

Kod testowy podzielono na osobne projekty, dostosowując je do warstwy architektury aplikacji. Projekt `Wellhicle.Domain.Tests` obejmuje testy reguł logiki domenowej, natomiast w ramach `Wellhicle.Application.Tests` przeprowadzone są testy klas obsługujących polecenia i zapytania, działające w pełnej izolacji od bazy danych. Projekt `Wellhicle.Infrastructure.Tests` obejmuje testy warstwy infrastruktury, weryfikując poprawność komunikacji z bazą danych oraz zewnętrznymi serwisami. Z kolei `Wellhicle.Presentation.Tests` obejmuje testy warstwy prezentacji, sprawdzając poprawność działania punktów końcowych API. W implementacji testów zastosowano dwa podstawowe atrybuty: „Fact” dla metod testowych bez parametrów oraz „Theory” z towarzyszącym „InlineData” dla metod parametryzowanych, uruchamianych wielokrotnie dla różnych zestawów danych wejściowych.

We wszystkich testach zastosowano wzorzec Arrange-Act-Assert (AAA), który dzieli treść metody testowej na trzy bloki: przygotowanie obiektów i danych wejściowych (Arrange), wywołanie testowanej metody (Act) oraz sprawdzenie rezultatu (Assert). Przyjęto również jednolitą konwencję nazewniczą „Metoda_Warunek_OczekiwanyRezultat", dzięki której nazwa metody odpowiada na trzy pytania: co jest testowane, w jakich warunkach oraz jaki jest oczekiwany efekt. Przykładowa nazwa: CalculateUpgradeCost_WhenUpgradingToMoreExpensivePlan_ShouldBePositive jednoznacznie opisuje weryfikowaną regułę biznesową. *Listing 89* przedstawia przykładowy test jednostkowy z warstwy domenowej, w którym wzorzec AAA jest wyraźnie widoczny.

Listing 89 Przykładowy test jednostkowy realizujący wzorzec AAA. Źródło: Opracowanie własne.

```
[Fact]
public void CalculateUpgradeCost_WhenUpgradingToMoreExpensivePlan_Shoul
dBePositive()
{
    //Arrange
    var start = new DateTime(2026, 1, 1, 0, 0, 0, DateTimeKind.Utc);
    var end = start.AddMonths(1);
    var upgradeAt = start.AddDays(10);

    //Act
    var cost = SubscriptionUpgradeCalculator.CalculateUpgradeCost(
        currentFullPrice: 3_000,
        newFullPrice: 6_000,
        periodStart: start,
        periodEnd: end,
        upgradeAt: upgradeAt);

    //Assert
    Assert.True(cost > 0);
}
```

W bloku Arrange przygotowano okres rozliczeniowy oraz moment zmiany planu. W bloku Act wywołano metodę kalkulatora z danymi opisującymi tańszy obecny plan i droższy plan docelowy. W bloku Assert sprawdzono wyłącznie znak rezultatu, ponieważ test weryfikuje regułę biznesową – zmiana na droższy plan zawsze wiąże się z dopłatą, a nie konkretną wartość liczbową, co czyni go odpornym na drobne modyfikacje formuły przy zachowaniu kierunku znaku.

Analogiczne podejście zastosowano w warstwie infrastruktury oraz warstwie prezentacji, gdzie testy weryfikują odpowiednio poprawność dostępu do danych oraz mapowanie wyników operacji na odpowiedzi HTTP. *Listing 90* przedstawia test, którego zadaniem jest wykrywanie nakładających się w czasie okresów użycia pojazdu.

Listing 90 Przykładowy test jednostkowy warstwy infrastruktury realizujący wzorzec AAA. Źródło: opracowanie własne.

```
[Fact(DisplayName = "INF-VUSE-02: wykrywanie nakadajcych si□kres□ pobranie listy kolizji")]
public async Task INF_VUSE_02_ShouldDetectOverlappingUsages_AndReturnConflicts()
{
    //Arrange
    using var context = CreateDbContext();
    var seed = await SeedUsageAggregateAsync(context, "Main", 1);
    var repository = new VehicleUsageReadRepository(context);

    var usage1 = CreateVehicleUsage(
        seed.Tenant.Id,
        seed.Assignment.Id,
        FixedUtcNow.AddHours(-2),
        FixedUtcNow.AddHours(2),
        VehicleUsageType.RESERVATION);
    var usage2 = CreateVehicleUsage(
        seed.Tenant.Id,
        seed.Assignment.Id,
        FixedUtcNow.AddHours(1),
        FixedUtcNow.AddHours(4),
        VehicleUsageType.MANUAL_BLOCK);

    context.VehicleUsages.AddRange(usage1, usage2);
    await context.SaveChangesAsync();

    //Act
    var hasConflict = await repository.AnyConflictingUsageAsync(
        seed.Tenant.Id,
        seed.Assignment.Id,
        usage1.DateFrom,
        usage1.DateTo,
        excludeUsageIds: new[] { usage1.Id });

    var conflicts = await repository.GetConflictingUsagesAsync(
        seed.Tenant.Id,
        seed.Assignment.Id,
        usage1.DateFrom,
        usage1.DateTo,
        excludeUsageIds: new[] { usage1.Id });

    //Assert
    Assert.True(hasConflict);
    Assert.Single(conflicts);
    Assert.Equal(usage2.Id, conflicts.Single().Id);
}
```

W bloku Arrange utworzono kontekst bazy danych, z minimalnym zestawem danych (najemca, pojazd wraz z przypisaniem) oraz zapisano dwa nakładające się w czasie okresy użycia pojazdu. W bloku Act zostały wywołane metody repozytorium, natomiast w bloku Assert sprawdzono, że kolizja została wykryta oraz że zwrócony został właśnie drugi, nakładający się okres. Test potwierdza, że zapytanie repozytorium poprawnie odwzorowuje regułę przedziałów czasowych w komunikacji z bazą danych, co jest podstawą kontroli dostępności pojazdów.

W warstwie prezentacji testy weryfikują między innymi poprawność mapowania kodów błędów na odpowiedzi HTTP. *Listing 91* przedstawia test parametryzowany, który dodatkowo stanowi przykład zastosowania atrybutu „Theory” z „InlineData”.

Listing 91 Przykładowy test jednostkowy warstwy prezentacji z atrybutem Theory. Źródło: opracowanie własne.

```
[Theory]
[InlineData(ErrorCode.NO_ACTIVE_SUBSCRIPTION)]
[InlineData(ErrorCode.PLAN_LIMIT_VEHICLES_EXCEEDED)]
[InlineData(ErrorCode.PLAN_LIMIT_DEPARTMENTS_EXCEEDED)]
[InlineData(ErrorCode.PLAN_LIMIT_OFFICE_WORKERS_EXCEEDED)]
[InlineData(ErrorCode.PLAN_LIMIT_OPERATIONAL_WORKERS_EXCEEDED)]
public void
ToActionResult_WhenSubscriptionLimitErrorCodesAreReturned_ReturnsForbidde
n(ErrorCode errorCode)
{
    //Arrange
    var controller = CreateController();
    var result = Result<object>.Failure("Subscription limit error.",
errorCode);

    //Act
    var actionResult = controller.ToActionResult(result);

    //Assert
    var objectResult = Assert.IsType<ObjectResult>(actionResult);
    var error = Assert.IsType<ErrorDto>(objectResult.Value);

    Assert.Equal(StatusCodes.Status403Forbidden,
objectResult.StatusCode);
    Assert.Equal((int)errorCode, error.Code);
}
```

W bloku Arrange przygotowano kontroler oraz wynik operacji reprezentujący błąd o zadanym kodzie. W bloku Act wywołano metodę rozszerzającą odpowiedzialną za mapowanie wyniku na odpowiedź HTTP. W bloku Assert sprawdzono kod statusu 403 oraz numer błędu w treści odpowiedzi. Dzięki atrybutowi „Theory" pojedyncza metoda testowa pokrywa pięć kodów błędów, co ogranicza powielanie kodu testowego.

Zastosowanie testów jednostkowych w projekcie Wellhicle przyczyniło się do poprawy stabilności warstwy serwerowej systemu oraz zwiększenia pewności zespołu przy wprowadzaniu zmian w kodzie. Szerokie pokrycie wszystkich warstw architektury aplikacji, od logiki domenowej przez warstwę aplikacji i infrastruktury aż po punkty końcowe API pozwoliło na wczesne wykrywanie błędów na każdym poziomie części serwerowej.

9. Problemy związane z systemem

W trakcie realizacji systemu Wellhicle pojawiły się trudności różnej natury. Część z nich ma charakter projektowy i wynika z okoliczności przedsięwzięcia, część natomiast dotyczy samego systemu oraz jego warstw technicznych. Rozdział opisuje te problemy wraz z ich zakresem i konsekwencjami.

9.1. Brak realnego zleceniodawcy

Projekt nie był realizowany dla zewnętrznego zleceniodawcy ani dla rzeczywistych użytkowników produkcyjnych. Zakres funkcjonalny i wymagania zespół określił samodzielnie, na podstawie analizy rynku i własnych założeń, bez możliwości skonfrontowania ich z oczekiwaniami konkretnej wypożyczalni. Miało to kilka konsekwencji. Wymagań nie zweryfikowano w praktyce, a priorytety ustalano na podstawie wewnętrznych przekonań. Część decyzji projektowych, jak rozgraniczenie odpowiedzialności między wypożyczalnią a oddziałem, okazała się niejednoznaczna dopiero w toku implementacji. Również testy prowadzono na danych przygotowanych przez zespół, a nie na rzeczywistym ruchu, co ograniczało wykrywanie błędów widocznych dopiero przy faktycznym użyciu.

9.2. Nieskuteczne unieważnianie tokenów sesji

Unieważnianie sesji oparto na liście unieważnionych tokenów przechowywanej w pamięci pojedynczej instancji serwera, bez trwałego, współdzielonego magazynu. Sesję opisują przy tym dwa rodzaje tokenów: krótkotrwały token dostępu oraz długotrwały token odświeżania, za pomocą którego uzyskuje się kolejne tokeny dostępu. Pojedyncze wylogowanie działało poprawnie, jednak nie prowadzono rejestru tokenów odświeżania wydanych danemu użytkownikowi. W rezultacie operacje, które miały zakończyć wszystkie sesje, nie unieważniały istniejących tokenów odświeżania. Dotyczyło to między innymi wylogowania ze wszystkich urządzeń, zmiany lub resetu hasła, dezaktywacji konta oraz usunięcia pracownika lub klienta. Część przypadków łagodziła kontrola stanu konta przy odświeżaniu, ponieważ dla konta dezaktywowanego lub usuniętego kolejna próba i tak była odrzucana. Najpoważniejsze w skutkach były zmiana lub reset hasła oraz wylogowanie ze wszystkich urządzeń, czyli typowa reakcja na podejrzenie przejęcia konta. Po tych operacjach wcześniej wydany token odświeżania zachowywał ważność aż do naturalnego wygaśnięcia. Lista unieważnionych tokenów nie była przy tym współdzielona między replikami serwera ani odporna na restart, więc po zwielokrotnieniu instancji unieważnienie nie obejmowało całego systemu.

9.3. Złączenie i rozproszenie ustawień regionalnych

Język potraktowano w systemie jako ogólne ustawienie regionalne. Ten sam parametr służył zarówno do tłumaczeń treści, jak i do formatowania danych, na przykład dat. Tymczasem język określa treść komunikatów i etykiet, a ustawienia regionalne wpływają na sposób prezentacji dat, liczb i walut. Połączenie obu w jednym parametrze sprawiało, że system nie rozróżniał tłumaczenia interfejsu od lokalnego sposobu prezentacji danych.

Oprócz tego te same cechy regionalne rozmieszczono na różnych poziomach modelu: strefę czasową traktowano jako cechę oddziału, a język i walutę jako cechy całej wypożyczalni. Komplikowało to interpretację danych w wypożyczalniach z oddziałami w różnych lokalizacjach: oddział mógł działać w innej strefie czasowej, mimo że język i walutę ustalano globalnie.

9.4. Niespójność kontraktów API między backendem a frontendami

Problemem była zmienność kontraktów API i jej wpływ na aplikacje klienckie. W części systemu pisanej w tej samej technologii co serwer kontrakt jest dostarczany jako klient generowany automatycznie z opisu API, więc zmiany propagują się bez ręcznej pracy. Trzy aplikacje przeglądarkowe konsumują API przez ręcznie utrzymywane usługi i typy danych, ponieważ po ich stronie nie ma generowanego klienta. Każda zmiana kontraktu konkretnego zasobu wymuszała więc równoległe poprawki w każdej z tych aplikacji. Biblioteka elementów powszechnie używanych w aplikacjach ujednolicała część przekrojową, taką jak paginacja, sortowanie i słowniki. Niespójność była najbardziej odczuwalna w podstawowych ścieżkach użytkownika, takich jak logowanie, rejestracja czy operacje na danych, stanowiących wejście dla wielu procesów.

9.5. Niekompletne trwałe usunięcie danych wypożyczalni

Trwałe usunięcie instancji wypożyczalni jest możliwe dopiero po jej dezaktywacji, ale faktyczne skasowanie powiązanych danych pozostaje niekompletne. Więzy integralności chronią część rekordów, takich jak pojazdy oraz dokumenty rozliczeniowe i płatności, blokując usunięcie wypożyczalni posiadającej takie dane. Konta użytkowników z kolei nie są powiązane bezpośrednio z wypożyczalnią i zostają w systemie jako osierocone. W efekcie operacja zapowiadana jako usunięcie wszystkich powiązanych danych nie wykonuje się w całości: dla niepustej wypożyczalni kończy się błędem integralności albo pozostawia rekordy bez właściciela.

9.6. Wieloplatformowość interfejsu mobilnego

Aplikacja mobilna powstała w technologii wieloplatformowej, ze wspólnym kodem dla systemów Android i iOS. Funkcje zależne od konkretnego urządzenia, jak dostęp do aparatu czy plików, wymagały pogodzenia różnic między platformami. Rozwiązano to warstwą abstrakcji oddzielającą wspólną logikę od implementacji właściwych dla każdej platformy, choć utrzymanie spójnego zachowania po obu stronach wymagało dodatkowej pracy.

9.7. Rozmiar obrazów Docker i limit przesyłania

Na etapie wdrażania problemem był rozmiar obrazów dockerowych. Limit przesyłania po stronie usługi brzegowej Cloudflare, rzędu 100 MB, blokował publikację większych warstw obrazu do rejestru kontenerowego, mimo że sam obraz budował się poprawnie. Ograniczenie pochodziło z warstwy brzegowej, a nie z własnego serwera pośredniczącego, który dopuszczał znacznie większe żądania. Wpływało to wprost na wdrożenie, ponieważ zbudowanie obrazu nie oznaczało, że można go opublikować.

9.8. Brak atomowości zapisu plików

Zapis pliku do magazynu obiektowego oraz zapis jego metadanych w bazie danych nie są wykonywane w ramach jednej transakcji. Gdy jedna z tych operacji się nie powiedzie, może powstać obiekt osierocony: plik bez odpowiadającego mu rekordu albo rekord bez pliku. Ten sam problem dotyczy usuwania. Jest to ograniczenie podobne do ryzyka utraty zdarzenia w kolejce komunikatów i wynika z łączenia dwóch niezależnych systemów trwałości bez mechanizmu kompensacji.

10. Plany na przyszłość

Część rozwiązań przyjętych w systemie celowo uproszczono ze względu na ograniczenia czasu i zasobów. Kompletna wersja systemu wymagałaby rozbudowy funkcji lub ich nowej implementacji. Kolejne podrozdziały przedstawiają kierunki dalszego rozwoju platformy.

10.1. Integracja płatności online

W modelu płatności typ do obsługi przez Stripe występuje jedynie jako znacznik typu płatności w danych, bez rzeczywistej integracji. Wybranie go nie uruchamia komunikacji z zewnętrzną bramką płatności. Następnym krokiem jest podłączenie oficjalnej biblioteki klienckiej Stripe oraz obsługa pełnego przepływu: inicjacji sesji płatności, odbioru potwierdzeń o statusie transakcji i automatycznych zwrotów. Dotyczy to zarówno rezerwacji, jak i zakupu subskrypcji. Pozwoli to domknąć w pełni online ścieżkę rezerwacja–płatność–potwierdzenie, bez ręcznej weryfikacji wpłat przez pracownika biurowego.

10.2. Wzmocnienie gwarancji dostarczania zdarzeń

Publikacja zdarzeń domenowych następuje po zatwierdzeniu transakcji bazodanowej i jest opakowana tak, aby awaria brokera wiadomości nie wycofywała zapisanych danych. W efekcie istnieje jednak ryzyko utraty pojedynczego zdarzenia, gdy zapis do bazy się powiódł, a publikacja do kolejki się nie powiodła. Następnym krokiem jest wprowadzenie wzorca *transactional outbox*. Polega on na zapisywaniu zdarzeń w tej samej transakcji co dane biznesowe i przekazywaniu ich do kolejki przez osobny proces, co eliminuje rozbieżność między stanem bazy a treścią kolejki i gwarantuje, że żadne zdarzenie nie zostanie pominięte [114].

10.3. Ograniczanie liczby żądań

Mechanizm ograniczania liczby żądań jest przygotowany w warstwie serwerowej, lecz nie został włączony. Przed publicznym wystawieniem systemu należy go włączyć i skonfigurować, w szczególności dla punktów końcowych dostępnych anonimowo, takich jak logowanie i odzyskiwanie hasła, gdzie ryzyko ataków *brute force* jest największe.

10.4. Wersjonowanie API

API nie jest obecnie jawnie wersjonowane. Wszystkie punkty końcowe udostępniono w ramach jednej, niepodzielonej według wersji przestrzeni ścieżek. Na obecnym etapie projektu, w którym jedynymi konsumentami są aplikacje rozwijane przez ten sam zespół, jest to akceptowalne. Wraz z planowanym otwarciem API na klientów zewnętrznych lub koniecznością obsługi starszych wersji aplikacji mobilnej wymagane będzie wprowadzenie schematu wersjonowania. Powinno zostać to zrobione przez przedrostek ścieżki lub nagłówek żądania, tak aby każda wersja API miała własny, niezależny pakiet kliencki.

10.5. Powiadomienia w czasie rzeczywistym

W warstwie serwerowej obsługa powiadomień w czasie rzeczywistym jest częściowo przygotowana. Zarejestrowano odpowiedni mechanizm oraz dodano komponent współdzielonego kanału między instancjami, oparty na Redisie. Nie został on jednak dokończony: brakuje aktywnego punktu odbioru po stronie serwera oraz klienta w aplikacjach przeglądarkowych i mobilnej, dlatego jedynym faktycznie dostarczanym kanałem komunikacji jest obecnie asynchroniczna poczta elektroniczna. Następnym krokiem jest dokończenie i włączenie tego mechanizmu, co pozwoli obsłużyć scenariusze wymagające natychmiastowej reakcji takie jak: aktualizację statusu wydania lub zwrotu pojazdu, powiadomienia operacyjne dla pracowników terenowych czy potwierdzenie opłacenia rezerwacji. Przy wdrożeniu na wielu instancjach niezbędne będzie pełne skonfigurowanie współdzielonego kanału, aby komunikat wysłany przez jedną instancję docierał do odbiorców podłączonych do pozostałych.

10.6. Logowanie przez zewnętrznego dostawcę

Uwierzytelnianie opiera się obecnie na własnym mechanizmie tokenów z logowaniem za pomocą hasła. Logowanie przez zewnętrznego dostawcę tożsamości, na przykład konto Google, nie jest zaimplementowane. Jego dodanie uprościłoby rejestrację klientów wypożyczalni i pozwoliło skorzystać z zabezpieczeń dostawcy zewnętrznego, takich jak wykrywanie podejrzanej aktywności czy uwierzytelnianie dwuetapowe. Wymagałoby to uzupełnienia konfiguracji uwierzytelniania o dodatkowy schemat oraz ścieżki rejestracji, w której zamiast weryfikacji hasła następuje walidacja tokenu dostawcy.

10.7. Publiczne API dla systemów ERP

Integracja z systemem ERP enova365 korzysta obecnie z wewnętrznego interfejsu programistycznego platformy: dodatek konsumuje te same punkty końcowe co aplikacje klienckie i uwierzytelnia się tym samym tokenem co użytkownicy. Choć interfejs udostępnia już funkcje przygotowane pod współpracę z ERP takie jak status rozliczeń po stronie ERP oraz przesyłanie dokumentów rozliczeniowych, to jednak nie istnieje wydzielona, publiczna warstwa API dla zewnętrznych systemów ERP. Następnym krokiem jest oddzielenie stabilnego, wersjonowanego API dla systemów tej klasy od interfejsu wewnętrznego oraz wprowadzenie uwierzytelniania *Machine to Machine* wraz z ograniczaniem liczby żądań w przeliczeniu na konsumenta.

10.8. Odtwarzalność danych wypożyczalni

System nie prowadzi automatycznych kopii zapasowych, a usunięcie rekordu jest operacją trwałą, bez możliwości jego późniejszego odtworzenia. Trwałość danych opiera się wyłącznie na wolumenie bazy danych. Następnym krokiem jest wprowadzenie regularnych kopii zapasowych przechowywanych przez określony czas oraz odtwarzania do punktu w czasie, a także mechanizmu eksportu i odtworzenia danych pojedynczej wypożyczalni. Uzupełnieniem byłoby opóźnione usuwanie, w którym dane przez wyznaczony czas pozostają możliwe do przywrócenia w razie pomyłki. Dzięki temu awaria lub pomyłka nie prowadziłaby już do nieodwracalnej utraty danych.

10.9. Globalny mechanizm obsługi wyjątków

W systemie zaimplementowano globalny mechanizm obsługi wyjątków, który dla znanych, przewidzianych błędów zwraca spójny obiekt zawierający kod oraz komunikat. Poza jego zakresem pozostają natomiast wyjątki nieobsłużone jawnie, które kończą się domyślną odpowiedzią o kodzie 500. Rozszerzenie mechanizmu o uniwersalny komponent przechwytujący również takie wyjątki i opakowujący je w ten sam format odpowiedzi zostało odłożone jako obszar dalszego rozwoju. Pozwoliłoby to ujednolicić obsługę błędów w całym interfejsie programistycznym i uprościć ich prezentację w aplikacjach klienckich.

10.10. Statystyki wypożyczalni

Na obecnym etapie panel zarządzania wypożyczalnią udostępnia podstawowe widoki operacyjne, a rozbudowane raportowanie świadomie odłożono jako funkcję o niższym priorytecie. Następnym krokiem jest dodanie widoku z miarami działalności wypożyczalni takimi jak: wykorzystaniem floty, przychodami oraz liczbą i statusami rezerwacji, wraz z punktami końcowymi agregującymi te dane. Widok ten mógłby stanowić ekran główny aplikacji do zarządzania wypożyczalnią.

10.11. Testy wydajnościowe, obciążeniowe i bezpieczeństwa

Dotychczasowe testy obejmowały poprawność funkcjonalną na poziomie manualnym, integracyjnym, automatycznym i jednostkowym. Nie przeprowadzono natomiast systematycznych testów wydajnościowych, obciążeniowych ani bezpieczeństwa. Ich wprowadzenie pozwoliłoby zmierzyć zachowanie systemu pod obciążeniem, wyznaczyć granice skalowania oraz zweryfikować odporność na typowe zagrożenia, zanim system zostanie wystawiony publicznie.

10.12. Refaktoryzacja kodu

Część rozwiązań przyjęto pod presją czasu i zasobów, co pozostawiło w systemie dług techniczny obejmujący zarówno warstwę serwerową, jak i aplikacje przeglądarkowe. Jego uporządkowanie wymagałoby najpierw dokładnego zaplanowania zakresu refaktoryzacji, ponieważ dotyczy ona rozproszonych obszarów kodu i może naruszyć działające już funkcje. Następnym krokiem jest sporządzenie takiego planu i stopniowe wprowadzanie zmian, co zwiększyłoby spójność oraz łatwość utrzymania kodu i uprościłoby dalszy rozwój systemu.

11. Podsumowanie

Zaplanowana platforma powstała i działa. Postawiony cel – zbudowanie kompletnego, wielotenantowego systemu SaaS dla wypożyczalni pojazdów – został osiągnięty: wdrożony system łączy zaplecze operacyjne wypożyczalni z publiczną, możliwą do personalizacji stroną jej klientów i obsługuje wiele niezależnych wypożyczalni w obrębie jednej platformy. To właśnie ta kombinacja, której nie zapewniał żaden z przeanalizowanych systemów konkurencyjnych, stanowi główny rezultat pracy.

Na rezultat złożyły się współpracujące ze sobą komponenty – serwer aplikacyjny, zestaw aplikacji przeglądarkowych ze wspólną biblioteką, aplikacja mobilna oraz własna infrastruktura uruchomieniowa – których szczegółową realizację przedstawiono w rozdziale o implementacji. Istotne jest, że wszystkie te części działają jako jedna, wdrożona całość, a poprawność rozwiązania potwierdzono testami manualnymi, integracyjnymi, automatycznymi oraz jednostkowymi obejmującymi wszystkie warstwy aplikacji.

Doprowadzenie systemu o takiej skali do działającej całości wymagało spójnych decyzji na wielu poziomach: od modelu dziedziny i architektury, przez podział na niezależnie wdrażane aplikacje, po organizację pracy zespołu i narzędzia wspierające koordynację. Skala przedsięwzięcia wymagała przy tym utrzymywania spójności między częściami rozwijanymi równolegle przez różne osoby.

Nie wszystkie założone wymagania udało się zrealizować w pełnym zakresie. Zdecydowaną większość funkcji zaimplementowano, jednak część pozostała niedokończona lub wykonana częściowo – przede wszystkim pełna obsługa płatności online, którą wraz z innymi kierunkami rozbudowy ujęto w rozdziale o następnych krokach. Napotkane trudności i ograniczenia techniczne, wynikające ze skali systemu oraz równoległego utrzymywania kilku aplikacji klienckich, omówiono w rozdziale poświęconym problemom związanym z systemem. Projekt interfejsu użytkownika opracowano kierunkowo, bez pokrycia wszystkich ekranów.

System pozostaje otwarty na dalszy rozwój – od pełnej integracji płatności online, przez wzmocnienie mechanizmów komunikacji i bezpieczeństwa, po przygotowanie interfejsu programistycznego dla zewnętrznych integratorów. Kierunki te przedstawiono w rozdziale o następnych krokach.

Realizacja systemu o tej skali w zespole była przedsięwzięciem wymagającym i rozłożonym w czasie. Pozwoliła zmierzyć się z projektowaniem złożonej architektury, pracą nowoczesnym stosem technologicznym oraz koordynacją wielu osób wokół wspólnego, działającego produktu. Doprowadzenie dużego, wielomodułowego systemu do postaci spójnej, wdrożonej całości stanowi najważniejszy efekt pracy.

12. Dodatki

12.1. Wykorzystanie narzędzi sztucznej inteligencji

Narzędzia oparte na dużych modelach językowych (LLM) towarzyszyły projektowi Wellhicle na etapach projektowania interfejsu użytkownika, implementacji systemu oraz przygotowania dokumentacji. Rozdział opisuje zakres tego zastosowania, użyte narzędzia oraz przyjęte zasady weryfikacji wyników.

12.1.1. Zastosowane narzędzia

W projekcie korzystano z następujących narzędzi AI:

- **GitHub Copilot** – asystent programistyczny firmy Microsoft zintegrowany ze środowiskiem IDE (Visual Studio, IntelliJ IDEA, Android Studio). Copilot działał w trybie uzupełniania kodu w edytorze (ang. *code completion*) oraz jako Copilot Chat – interfejs konwersacyjny umożliwiający zadawanie pytań w kontekście otwartego pliku lub projektu. [149]
- **Cursor** – edytor kodu z wbudowanym asystentem AI, zbudowany na bazie środowiska Visual Studio Code. Wykorzystywany do uzupełniania i refaktoryzacji fragmentów kodu w pracy nad backendem. [150]
- **Claude Code** – narzędzie CLI firmy Anthropic uruchamiane lokalnie w katalogu roboczym projektu, oparte na modelach Claude Sonnet i Opus (zależnie od zadania). Praca w kontekście całego repozytorium pozwalała na analizę zależności między plikami oraz generowanie tekstu ściśle powiązanego z rzeczywistą strukturą systemu. [151]
- **Claude Cowork** – narzędzie firmy Anthropic do pracy z plikami i dokumentacją w katalogu roboczym, oparte na modelach Claude. Stosowane przy eksploracji materiałów projektowych oraz przy redakcji i przygotowywaniu wstępnych wersji fragmentów dokumentacji. [152]
- **Claude** w interfejsie przeglądarkowym – konwersacyjna wersja modeli Claude dostępna przez interfejs webowy. Wykorzystywana do wyjaśniania i analizy kodu – w szczególności przez członków zespołu, którzy dopiero zaznajamiali się z narzędziami AI. [153]
- **Google Gemini** – model językowy firmy Google dostępny przez interfejs webowy. Stosowany był przy projektowaniu interfejsu użytkownika – do generowania propozycji układu ekranów, doboru palety barw oraz oceny spójności wizualnej komponentów UI. [154]
- **ChatGPT** – model językowy firmy OpenAI dostępny przez interfejs webowy. W pracy nad backendem stosowany przede wszystkim w początkowej fazie projektu (wyjaśnianie fragmentów kodu, diagnozowanie błędów), a w dalszym toku prac zastąpiony innymi narzędziami. Stosowany ponadto przy redakcji tekstu dokumentacyjnego. [155]
- **Codex** – narzędzie firmy OpenAI ukierunkowane na generowanie i analizę kodu. Stosowane przy pracy z *backendem* oraz dokumentacją techniczną; wymagało starannej konfiguracji, gdyż nieodpowiednio dostrojone mogło generować sugestie niespójne z istniejącą strukturą projektu. [156]
- **Bielik (LM Studio)** – otwarto źródłowy model językowy uruchamiany lokalnie za pomocą aplikacji LM Studio, bez przesyłania treści do zewnętrznych serwerów. Stosowany do korekty językowej tekstów dokumentacyjnych bez zmiany treści merytorycznej, faktów ani struktury tekstu. [157] [158]

Poszczególne narzędzia były dobierane przez autorów w zależności od obszaru pracy: Gemini stosowano przy projektowaniu warstwy wizualnej, GitHub Copilot, Cursor, Codex i ChatGPT towarzyszyły pracy programistycznej nad *backendem*, GitHub Copilot i Claude Code – pracy nad *frontendem*, natomiast Claude Code, Claude Cowork i Bielik – pracy redakcyjnej nad dokumentacją.

12.1.2. Projektowanie interfejsu użytkownika

Google Gemini stosowany był na etapie projektowania warstwy wizualnej aplikacji przeglądarkowych. Narzędzie wspomagało generowanie propozycji układu ekranów, dobór typografii oraz ocenę czytelności i spójności komponentów UI. Umożliwiało szybkie iterowanie po wariantach projektu bez konieczności każdorazowego prototypowania ich w narzędziu graficznym. Ostateczny kształt interfejsu był każdorazowo weryfikowany przez autora odpowiedzialnego za projekt interfejsu i dostosowywany do wymagań funkcjonalnych systemu.

12.1.3. Zakres zastosowania w implementacji

Narzędzia AI wspierały pracę programistyczną w kilku obszarach: uzupełniania kodu, generowania odtwórczych fragmentów implementacji *backendu*, wsparcia przy integracji zewnętrznych systemów, tworzenia testów, przygotowania danych testowych, analizy istniejących fragmentów implementacji oraz jednorazowego przygotowania materiału do modelowania UML.

12.1.3.1. Uzupełnianie kodu i podpowiedzi składniowe

Przy pisaniu kodu stosowano różne narzędzia w zależności od obszaru systemu. W obszarze *frontendu* autouzupełniania w edytorze dostarczały GitHub Copilot i Claude Code – generowały podpowiedzi na podstawie nazw metod i komentarzy, a także obsługiwały powtarzalne fragmenty implementacji.

W praktyce zaobserwowano, że narzędzia AI generowały niekiedy nadmiernie rozbudowane rozwiązania: w sytuacjach, gdzie wystarczyłoby kilkanaście linii kodu, wynik zawierał wielokrotnie więcej fragmentów wymagających późniejszej refaktoryzacji. Funkcjonalność wygenerowanego kodu była poprawna, lecz jego zwięzłość i jakość wymagały ręcznej korekty. Z tego względu każda sugestia była oceniana przez autora przed akceptacją.

12.1.3.2. Generowanie kodu odtwórczego – backend

W obszarze *backendu* narzędzia AI wspierały tworzenie powtarzalnych i odtwórczych fragmentów kodu: obiektów transferu danych (DTO) i obiektów żądań, walidatorów `FluentValidation`, komentarzy dokumentacyjnych XML oraz konfiguracji dokumentacji API (OpenAPI, narzędzie Scalar). Pomoc obejmowała także prostsze metody klas domenowych, interceptory, helpery repozytoriów oraz fragmenty obsługujące integrację z zewnętrznym źródłem danych o pojazdach (vPIC). Znaczący udział narzędzi odnotowano w pracach nad modułem rozliczeń.

W roli asystentów programistycznych w tym obszarze stosowano GitHub Copilot, Cursor, Codex i ChatGPT; każdy wygenerowany fragment był oceniany, testowany i w razie potrzeby poprawiany przed włączeniem do repozytorium.

12.1.3.3. Integracja z systemem enova365

Przy integracji z systemem ERP enova365 narzędzia AI wspierały przygotowanie metod konwersji typów i mapowania danych, wstępny szkic warstwy pośredniczącej (adaptera) oraz obsługę błędów i logowanie. Sposób integracji i decyzje dotyczące jej kształtu podejmował autor; rola narzędzi ograniczała się do wsparcia przy kodzie odtwórczym.

12.1.3.4. Testy jednostkowe i integracyjne

Testy *backendu* – zarówno jednostkowe, jak i integracyjne – były generowane przez narzędzia AI. Zakres testów oraz oczekiwane zachowania systemu definiował zespół na podstawie wymagań i logiki domenowej, natomiast wygenerowany kod testów był każdorazowo weryfikowany przez uruchomienie oraz przegląd. Podejście to pozwoliło objąć testami ścieżki pozytywne, przypadki brzegowe i obsługę błędów przy zachowaniu kontroli nad poprawnością sprawdzanych warunków.

12.1.3.5. Dane testowe

Do wypełnienia środowiska danymi testowymi wykorzystano generator (ang. *seeder*) przygotowany z udziałem narzędzi AI. Stanowił on narzędzie deweloperskie wspomagające pracę i testy, pozostające poza zakresem dostarczanego produktu.

12.1.3.6. Wyjaśnianie i analiza kodu

Copilot Chat, ChatGPT i Codex stosowane były do analizy fragmentów kodu wymagających wyjaśnienia – szczególnie przy integracji z bibliotekami zewnętrznymi (m.in. *Ktor*, *Exposed*, *Koin*, *OpenXML*) lub przy pracy z kodem napisanym przez innego członka zespołu. Narzędzia generowały opis działania kodu w języku naturalnym, co skracało czas potrzebny na zrozumienie nowych obszarów projektu. ChatGPT stosowano ponadto do diagnozowania przyczyn błędów kompilacji i uruchomienia, szczególnie przy nieoczywistych komunikatach błędów bibliotek zewnętrznych.

12.1.3.7. Przygotowanie kodu domenowego pod import do Enterprise Architect

Do opracowania diagramów klas zastosowano narzędzie Sparx Enterprise Architect z funkcją inżynierii wstecznej. Podejście to wymaga dostarczenia kodu źródłowego do narzędzia modelującego, co w przypadku *backendu* Wellhicle nie było możliwe bez wstępnego oczyszczenia, co dokładniej zostało opisane w punkcie *4.1.4*.

W celu przygotowania materiału do importu Claude Code jednorazowo i automatycznie oczyścił roboczą kopię warstwy domenowej według ustalonego wzorca przekształcenia: usunął wymienione elementy techniczne oraz ujednolicił kodowanie plików pod polskie środowisko systemowe, tak aby do Enterprise Architect trafiły wyłącznie klasy domenowe wraz z atrybutami i metodami. Robocza kopia służyła wyłącznie do importu i celowo nie była kompilowalna.

Decyzje merytoryczne pozostały po stronie autorów: wybór wzorca przekształcenia, ręczne usunięcie wybranych klas (serwisy domenowe, klasy pomocnicze) oraz uzupełnienia wykraczające poza mechaniczne czyszczenie przeprowadzono bez udziału narzędzia. Rola AI ograniczała się do powtarzalnego, narzędziowego wykonania ustalonego zestawu operacji na kodzie.

12.1.3.8. Weryfikacja spójności diagramów

Na etapie tworzenia diagramów modelowania narzędzia AI stosowane były do weryfikacji spójności diagramu z towarzyszącym mu opisem tekstowym. Po przygotowaniu wstępnej wersji diagramu i opisu narzędzie wskazywało rozbieżności między reprezentacją graficzną a tekstem. Weryfikacja miała charakter pomocniczy – ocena spójności i decyzja o korekcie należały do autora odpowiedzialnego za daną część dokumentacji.

12.1.4. Zakres zastosowania w dokumentacji

W obszarze dokumentacji narzędzia oparte na modelach Claude (Claude Code, Claude Cowork) oraz Bielik pełniły rolę asystentów redakcyjnych.

12.1.4.1. Podsumowania rozwiązań technicznych

Claude Code i Claude Cowork stosowane były do przekształcania artefaktów inżynierskich – fragmentów kodu, schematów architektury, notatek ze spotkań projektowych – w opisy tekstowe na potrzeby dokumentacji. Narzędzia otrzymywały jako kontekst rzeczywisty kod z repozytorium i na tej podstawie generowały wstępne szkice sekcji opisujące dane rozwiązanie. Dobór treści, jej interpretacja oraz ostateczna redakcja pozostawały po stronie autorów.

12.1.4.2. Eksploracja i porządkowanie materiałów projektowych

Ze względu na obszerność materiałów zgromadzonych w trakcie projektu – około 89 nagranych spotkań, notatek oraz zgłoszeń w systemie Jira – narzędzia AI wykorzystywano do ich przeszukiwania i porządkowania. Wsparcie miało charakter nawigacyjny: ułatwiało odnalezienie i uporządkowanie informacji, nie było natomiast źródłem analiz ani wniosków, które pozostały po stronie autorów.

12.1.4.3. Redakcja i korekta językowa

Teksty przygotowywane przez autorów poddawano korekcie językowej z wykorzystaniem narzędzi AI – przede wszystkim modelu Bielik, który nie ingerował w treść merytoryczną, fakty ani strukturę tekstu. Claude Code i Claude Cowork stosowano natomiast w przypadkach wymagających pracy w bezpośrednim kontekście kodu źródłowego lub przy formułowaniu objaśnień działania konkretnych funkcji.

12.1.4.4. Strukturyzacja rozdziałów

Na etapie planowania rozdziałów Claude Code był używany do weryfikacji kompletności zaproponowanych struktur – czy wszystkie istotne aspekty implementacyjne zostały ujęte i czy kolejność podrozdziałów jest logiczna. Ostateczne decyzje dotyczące struktury i zawartości podejmował każdorazowo autor odpowiedzialnego rozdziału.

12.1.5. Weryfikacja wyników

Przyjętą przez autorów zasadą była weryfikacja każdego fragmentu wygenerowanego przez AI przed jego użyciem – zarówno kodu, jak i tekstu.

W przypadku kodu weryfikacja obejmowała:

- uruchomienie testów automatycznych i manualne sprawdzenie funkcjonalności,
- analizę logiki wygenerowanego fragmentu pod kątem zgodności z architekturą systemu,
- przegląd kodu przez innego członka zespołu w ramach procesu code review (pull request).

W przypadku dokumentacji weryfikacja obejmowała:

- porównanie faktów technicznych z kodem źródłowym repozytorium (wersje bibliotek, nazwy modułów, struktura klas),
- konfrontację opisów z oficjalną dokumentacją stosowanych technologii,
- merytoryczną ocenę przez autora odpowiedzialnego za dany obszar systemu,
- konsultacje z promotorem; źródłem prawdy pozostawał dokument redagowany ręcznie przez autorów.

Żaden fragment, ani kod, ani tekst, nie był włączany do projektu bez przejścia przez powyższy proces weryfikacji.

12.1.6. Wnioski

Zastosowanie narzędzi AI skróciło czas realizacji powtarzalnych zadań implementacyjnych i redakcyjnych. Kluczowym warunkiem uzyskania użytecznych wyników okazało się dostarczanie narzędziom precyzyjnego kontekstu. Modele generowały wartościowe odpowiedzi tylko wtedy, gdy otrzymywały rzeczywiste artefakty projektu (kod, schematy, specyfikacje), nie zaś ogólne opisy problemu.

W ciągu dwóch lat poprzedzających realizację projektu, narzędzia AI stały się standardowym elementem warsztatu inżyniera oprogramowania. Według badania „Stack Overflow Developer Survey" odsetek deweloperów używających tych narzędzi lub planujących ich wdrożenie wzrósł z 76% w 2024 r. do 84% w 2025 r. Codzienne użycie deklaruje już 47% respondentów [159].

Warte odnotowania jest towarzyszące temu zjawisku napięcie: mimo że w 2025 r. 84% deweloperów korzysta z narzędzi AI, zaufanie do ich wyników spadło do 29%, czyli o 11 punktów procentowych rok do roku[159] , [160]. Oznacza to, że branża wypracowuje model pracy, w którym AI przyspiesza wykonanie zadania, lecz odpowiedzialność za jego wynik pozostaje po stronie człowieka. Zasady weryfikacji przyjęte w projekcie Wellhicle, opisane w punkcie *12.1.5*, wpisują się w ten model.

W trakcie pracy ujawniły się także ograniczenia sygnalizowane we wcześniejszych podrozdziałach: skłonność narzędzi do generowania nadmiernie rozbudowanych rozwiązań oraz wrażliwość niektórych z nich na konfigurację. W obu przypadkach obowiązek oceny i korekty spoczywał po stronie autora.

Narzędzia AI nie zastępowały decyzji projektowych ani wiedzy domenowej autorów. Pełniły rolę asystentów przyspieszających realizację zadań, których zakres i kierunek były każdorazowo określane przez człowieka.

12.2. Słownik pojęć

- **Administrator platformy** – operator platformy Wellhicle zarządzający wypożyczalniami, pakietami, subskrypcjami i ustawieniami globalnymi; rola nadrzędna, odrębna od właściciela wypożyczalni, z możliwością przejęcia jego roli na potrzeby wsparcia technicznego.
- **Agregat** – w modelu domenowym spójna grupa obiektów modyfikowana jako całość; korzeń agregatu wyznacza granicę spójności i jest jedynym punktem zmiany jej stanu.
- **Aplikacja do administrowania wypożyczalniami** – aplikacja przeglądarkowa administratora platformy służąca do zarządzania wypożyczalniami i ustawieniami globalnymi.
- **Aplikacja do korzystania z usług wypożyczalni** – aplikacja udostępniające klientom wypożyczalni oferty, rejestrację, składanie rezerwacji oraz panel klienta.
- **Aplikacja do prezentacji oferty systemu** – aplikacja pełniące funkcję informacyjno-promocyjną platformy i stanowiące punkt startowy rejestracji nowych wypożyczalni.
- **Aplikacja do zarządzania wypożyczalniami** – główna aplikacja przeglądarkowa właściciela i pracowników biurowych, służąca do prowadzenia wypożyczalni.
- **Aplikacja mobilna** – wieloplatformowa aplikacja (Android, iOS) dla pracowników operacyjnych, obsługująca procesy terenowe: wydanie i zwrot pojazdu oraz zgłaszanie usterek.
- **Broker wiadomości** – komponent pośredniczący w komunikacji między serwerem aplikacyjnym a procesami roboczymi, umożliwiający asynchroniczne przetwarzanie zadań przez trwałe kolejki komunikatów (w projekcie RabbitMQ).
- **Biblioteka elementów współdzielonych pomiędzy aplikacjami** – biblioteka współdzieląca komponenty, typy danych i mechanizmy komunikacji z API między aplikacjami przeglądarkowymi, ograniczająca powielanie kodu.
- **Epik** – w systemie Jira nadrzędna grupa prac obejmująca moduł lub obszar funkcjonalny systemu.
- **Gość platformy SaaS** – niezalogowany odwiedzający stronę publiczną platformy; przegląda ofertę pakietów i może rozpocząć rejestrację wypożyczalni, po czym przechodzi w rolę właściciela. Nie jest tożsamy z gościem wypożyczalni.
- **Gość wypożyczalni** – niezalogowany użytkownik strony publicznej konkretnej wypożyczalni; przegląda i filtruje oferty oraz inicjuje rezerwację, lecz jej finalizacja wymaga logowania lub rejestracji.
- **Handler** – klasa obsługująca pojedynczą komendę lub zapytanie w warstwie aplikacyjnej (zgodnie z CQRS): wykonuje regułę przypadku użycia i zwraca wynik, podczas gdy kontrolery jedynie przekazują żądania do właściwego handlera.
- **Interceptor** – mechanizm przechwytujący operacje w celu dołączenia wspólnej logiki: w aplikacjach przeglądarkowych żądania HTTP (dołączanie tokenu, obsługa błędów), a w warstwie danych operacje zapisu Entity Framework Core (na przykład porządkowanie plików).
- **Klient platformy** – wypożyczalnia korzystająca z Wellhicle jako usługi do prowadzenia własnej działalności; podmiot wykupujący dostęp do platformy. Odróżniany od klienta wypożyczalni.
- **Klient SDK API (Wellhicle.Client)** – typowany pakiet kliencki generowany z kontraktu OpenAPI (narzędziem Kiota), upraszczający budowę integracji z API platformy; jego głównym odbiorcą jest integracja z systemem ERP.
- **Klient wypożyczalni** – zarejestrowany użytkownik strony publicznej wypożyczalni, który rezerwuje i wynajmuje pojazdy, podpisuje elektronicznie umowę najmu i prowadzi historię

rezerwacji; danymi oraz kontami klientów zarządza moduł Klienci. Odróżniany od klienta platformy.

- **Korzeń agregatu** – encja stanowiąca jedyny punkt dostępu do agregatu i gwarant jego spójności; przez nią przechodzą wszystkie modyfikacje stanu agregatu.
- **Kreator wypożyczalni** – prowadzony proces pierwszej konfiguracji nowej instancji, uruchamiany przy pierwszym logowaniu właściciela i warunkujący dalszą pracę w panelu.
- **Obiekt wartości** – element modelu domenowego identyfikowany przez zestaw składowych, a nie własną tożsamość (na przykład adres e-mail lub numer telefonu), niedopuszczający stanu niepoprawnego.
- **Obserwowalność (ang. *observability*)** – zestaw mechanizmów logowania, metryk i śledzenia działania systemu, oparty na instrumentacji OpenTelemetry oraz stosie Prometheus, Loki, Tempo i Grafana.
- **Pracownik biurowy** – użytkownik panelu zarządzania wypożyczalnią obsługujący flotę, klientów, oferty i rezerwacje oraz prowadzący rozliczenia sprzedażowe.
- **Pracownik operacyjny** – użytkownik aplikacji mobilnej realizujący wydanie i odbiór pojazdu oraz zgłaszanie usterek; jego zakres jest wyraźnie oddzielony od zakresu pracownika biurowego.
- **Unit of Work** – wzorzec grupujący zmiany na obiektach domenowych w jedną transakcję i zatwierdzający je wspólnie, co zapewnia spójność zapisu; w projekcie współpracuje z repozytoriami i kontekstem danych.
- **Webhook** – mechanizm automatycznego powiadamiania jednej aplikacji przez drugą o wystąpieniu zdarzenia za pomocą żądania HTTP; w projekcie wykorzystany między innymi do przekazywania powiadomień z systemu Jira na kanał Discord.
- **Wielojęzyczność** – przystosowanie systemu do obsługi wielu języków; zakres wsparcia językowego różni się między aplikacjami, najszerszy w aplikacjach klienckich.
- **Wielotenantowość** – cecha systemu, w którym wiele wypożyczalni korzysta ze wspólnej instancji bazy danych przy logicznej izolacji danych po identyfikatorze najemcy.
- **Wielowalutowość** – obsługa wielu walut (w bieżącej konfiguracji trzynastu) w cennikach ofert, rozliczeniach i płatnościach.
- **Właściciel wypożyczalni** – osoba reprezentująca firmę i zarządzająca jedną wypożyczalnią; dziedziczy uprawnienia pracownika biurowego i operacyjnego oraz zarządza subskrypcją i konfiguracją instancji. Rola odrębna od administratora platformy.
- **Zdarzenie domenowe** – niemodyfikowalny zapis faktu zaistniałego w modelu domenowym, publikowany po pomyślnym utrwaleniu zmian i wykorzystywany do reakcji w innych częściach systemu.

12.3. Spis tabel

Tabela 1 Zestawienie istniejących rozwiązań. Źródło: opracowanie własne. 20
Tabela 2 Podział odpowiedzialności w projekcie Wellhicle. Źródło: opracowanie własne. 23
Tabela 3 Zakres obowiązków w zespole analitycznym. Źródło: opracowanie własne. 24
Tabela 4 Zakres obowiązków w zespole projektowania systemu i architektury. Źródło: opracowanie własne. 24
Tabela 5 Zakres obowiązków w zespole backendowym. Źródło: opracowanie własne. 25
Tabela 6 Zakres obowiązków w zespole frontendowym. Źródło: opracowanie własne. 25

Tabela 7 Zakres obowiązków w zespole testowania. Źródło: opracowanie własne. 26
Tabela 8 Kanały komunikacji i ich typowe zastosowanie w projekcie. Źródło: opracowanie własne. 34
Tabela 9 Porównanie planu pierwotnego z rzeczywistym przebiegiem realizacji. Źródło: opracowanie własne. 38
Tabela 10 Rodzaje artefaktów analizy i ich oznaczenia. Źródło: opracowanie własne. 60
Tabela 11 Moduły funkcjonalne systemu i ich skróty. Źródło: opracowanie własne..... 60
Tabela 12 Podsumowanie wszystkich aktorów systemu. Źródło: opracowanie własne. 63
Tabela 13 Aktorzy modułu Zarządzanie wypożyczalnią. Źródło: opracowanie własne. 67
Tabela 14 Reguły biznesowe i walidacje modułu Zarządzanie wypożyczalnią. Źródło: opracowanie własne. 69
Tabela 15 Powiązania modułu Zarządzanie wypożyczalnią z innymi modułami systemu. Źródło: opracowanie własne. 70
Tabela 16 Aktorzy modułu Flota. Źródło: opracowanie własne. 70
Tabela 17 Reguły biznesowe i walidacje modułu Flota. Źródło: opracowanie własne. 71
Tabela 18 Powiązania modułu Flota z innymi modułami systemu. Źródło: opracowanie własne..... 72
Tabela 19 Aktorzy modułu Klienci. Źródło: opracowanie własne. 73
Tabela 20 Reguły biznesowe i walidacje modułu Klienci. Źródło: opracowanie własne. 74
Tabela 21 Powiązania modułu Klienci z innymi modułami systemu. Źródło: opracowanie własne. ... 74
Tabela 22 Aktorzy modułu Oferty. Źródło: opracowanie własne. 75
Tabela 23 Reguły biznesowe i walidacje modułu Oferty. Źródło: opracowanie własne..... 76
Tabela 24 Powiązania modułu Oferty z innymi modułami systemu. Źródło: opracowanie własne. 77
Tabela 25 Aktorzy modułu Rezerwacje. Źródło: opracowanie własne. 77
Tabela 26 Reguły biznesowe i walidacje modułu Rezerwacje. Źródło: opracowanie własne. 80
Tabela 27 Powiązania modułu Rezerwacje z innymi modułami systemu. Źródło: opracowanie własne. 81
Tabela 28 Aktorzy modułu Finanse. Źródło: opracowanie własne. 82
Tabela 29 Reguły biznesowe i walidacje modułu Finanse. Źródło: opracowanie własne. 85
Tabela 30 Powiązania modułu Finanse z innymi modułami systemu. Źródło: opracowanie własne. .. 86
Tabela 31 Aktorzy modułu Płatności. Źródło: opracowanie własne. 87
Tabela 32 Reguły biznesowe i walidacje modułu Płatności. Źródło: opracowanie własne. 89
Tabela 33 Powiązania modułu Płatności z innymi modułami systemu. Źródło: opracowanie własne. 90
Tabela 34 Aktorzy modułu Operacyjny. Źródło: opracowanie własne. 90
Tabela 35 Reguły biznesowe i walidacje modułu Operacyjny. Źródło: opracowanie własne. 91
Tabela 36 Powiązania modułu Operacyjny z innymi modułami systemu. Źródło: opracowanie własne. 92
Tabela 37 Aktorzy modułu Strona publiczna wypożyczalni. Źródło: opracowanie własne..... 93
Tabela 38 Reguły biznesowe i walidacje modułu Strona publiczna wypożyczalni. Źródło: opracowanie własne. 94
Tabela 39 Powiązania modułu Strona publiczna wypożyczalni z innymi modułami systemu. Źródło: opracowanie własne. 94
Tabela 40 Aktorzy modułu Pakiety. Źródło: opracowanie własne. 95
Tabela 41 Reguły biznesowe i walidacje modułu Pakiety. Źródło: opracowanie własne. 96
Tabela 42 Powiązania modułu Pakiety z innymi modułami systemu. Źródło: opracowanie własne. ... 96
Tabela 43 Aktorzy modułu Administracja platformy. Źródło: opracowanie własne. 97
Tabela 44 Reguły biznesowe i walidacje modułu Administracja platformy. Źródło: opracowanie własne. 99

Tabela 45 Powiązania modułu Administracja platformy z innymi modułami systemu. Źródło: opracowanie własne. 99
Tabela 46 Aktorzy modułu Komunikacja. Źródło: opracowanie własne. 100
Tabela 47 Reguły biznesowe i walidacje modułu Komunikacja. Źródło: opracowanie własne. 101
Tabela 48 Powiązania modułu Komunikacja z innymi modułami systemu. Źródło: opracowanie własne. 101
Tabela 49 Aktorzy modułu Strona publiczna platformy. Źródło: opracowanie własne. 102
Tabela 50 Reguły biznesowe i walidacje modułu Strona publiczna platformy. Źródło: opracowanie własne. 102
Tabela 51 Powiązania modułu Strona publiczna platformy z innymi modułami systemu. Źródło: opracowanie własne. 103
Tabela 52 Scenariusz przypadku użycia – anonimizacja klienta. Źródło: opracowanie własne. 132
Tabela 53 Scenariusz przypadku użycia – przeglądanie historii rezerwacji. Źródło: opracowanie własne. 134
Tabela 54 Scenariusz przypadku użycia – tworzenie rezerwacji w imieniu klienta. Źródło: opracowanie własne. 136
Tabela 55 Scenariusz przypadku użycia – wydanie pojazdu. Źródło: opracowanie własne. 138
Tabela 56 Scenariusz przypadków użycia – odbiór pojazdu. Źródło: opracowanie własne. 141
Tabela 57 Scenariusz przypadku użycia – zgłoszenie usterki. Źródło: opracowanie własne. 143
Tabela 58 Scenariusz przypadku użycia – zmiana własnej domeny. Źródło: opracowanie własne. 144
Tabela 59 Zestawienie adresów dostępnych w systemie Wellhicle oraz ich przeznaczenie. Źródło: opracowanie własne. 306
Tabela 60 Zestawienie głównych przepływów integracji kodu. Źródło: opracowanie własne. 310
Tabela 61 Powiązanie pojęć platformy Wellhicle z encjami systemu enova365. Źródło: opracowanie własne. 316
Tabela 62 Fragment raportu testów manualnych strony publicznej wypożyczalni. Źródło: opracowanie własne. 322

12.4. Spis rysunków

Rysunek 1 Panel operacyjny RentSyst z widokiem floty. Źródło: opracowanie własne. 15
Rysunek 2 Widok konfiguracji widżetów RentSyst do integracji z własną stroną. Źródło: opracowanie własne. 16
Rysunek 3 Fragment panelu operacyjnego HQ Rental Software z widokiem floty. Źródło: opracowanie własne. 17
Rysunek 4 Widok aktywnych odbiorów w aplikacji mobilnej HQ Rental Software. Źródło: opracowanie własne. 17
Rysunek 5 Panel operacyjny RentHelp.io z widokiem floty. Źródło: opracowanie własne. 18
Rysunek 6 Widżet rezerwacji w RentHelp.io, do osadzenia na własnej stronie internetowej. Źródło: opracowanie własne. 19
Rysunek 7 Panel operacyjny FleetWire.io z widokiem rezerwacji. Źródło: opracowanie własne..... 19
Rysunek 8 Widok domyślnej strony internetowej FleetWire.io. Źródło: opracowanie własne. 20
Rysunek 9 Struktura projektów Jira z podziałem na tablice zespołów. Źródło: opracowanie własne. . 28
Rysunek 10 Przepływ statusów zgłoszeń w projekcie „WELL". Źródło: opracowanie własne. 29
Rysunek 11 Struktura przestrzeni dokumentacyjnej „DOCS" w Confluence. Źródło: opracowanie własne. 31
Rysunek 12 Przepływ statusów zgłoszeń w projekcie „WD". Źródło: opracowanie własne..... 36
Rysunek 13 Widok startowy Scalar. Źródło: opracowanie własne. 56
Rysunek 14 Widok pojedynczego punktu końcowego API. Źródło: opracowanie własne. 57
Rysunek 15 Efekt końcowy implementacji autorskiego rozwiązania w Scalar. Źródło: opracowanie własne. 58
Rysunek 16 Hierarchia dziedziczenia aktorów systemu Wellhicle. Źródło: opracowanie własne..... 66
Rysunek 17 Diagram przypadków użycia modułu – zarządzanie wypożyczalnią. Źródło: opracowanie własne. 107
Rysunek 18 Diagram przypadków użycia modułu – flota. Źródło: opracowanie własne. 108
Rysunek 19 Diagram przypadków użycia modułu – klienci. Źródło: opracowanie własne. 109
Rysunek 20 Diagram przypadków użycia modułu – oferty. Źródło: opracowanie własne. 110
Rysunek 21 Diagram przypadków użycia modułu – rezerwacje. Źródło: opracowanie własne. 111
Rysunek 22 Diagram przypadków użycia modułu – płatności. Źródło: opracowanie własne. 112
Rysunek 23 Diagram przypadków użycia modułu – operacyjny. Źródło: opracowanie własne..... 113
Rysunek 24 Diagram przypadków użycia modułu – strona publiczna wypożyczalni. Źródło: opracowanie własne. 114
Rysunek 25 Diagram przypadków użycia modułu – pakiety. Źródło: opracowanie własne..... 115
Rysunek 26 Diagram przypadków użycia modułu – administracja platformy. Źródło: opracowanie własne. 116
Rysunek 27 Diagram przypadków użycia modułu – komunikacja. Źródło: opracowanie własne..... 117
Rysunek 28 Diagram przypadków użycia modułu – strona publiczna platformy. Źródło: opracowanie własne. 118
Rysunek 29 Diagram potwierdzenia wykonania. Źródło: opracowanie własne. 119
Rysunek 30 Diagram komunikatu potwierdzenia. Źródło: opracowanie własne. 120
Rysunek 31 Diagram komunikatu odrzucenia. Źródło: opracowanie własne. 120
Rysunek 32 Diagram odpowiedzi systemu. Źródło: opracowanie własne. 121
Rysunek 33 Diagram wypełnienia formularza. Źródło: opracowanie własne. 122
Rysunek 34 Diagram weryfikacji warunków akcji. Źródło: opracowanie własne..... 123
Rysunek 35 Diagram zarządzania obiektem. Źródło: opracowanie własne. 124

Rysunek 36 Diagram wyświetlenia szczegółów obiektu. Źródło: opracowanie własne. 125
Rysunek 37 Diagram edycji obiektu. Źródło: opracowanie własne. 126
Rysunek 38 Diagram dodania obiektu. Źródło: opracowanie własne. 127
Rysunek 39 Diagram usunięcia obiektu. Źródło: opracowanie własne. 128
Rysunek 40 Diagram aktywacji obiektu. Źródło: opracowanie własne. 129
Rysunek 41 Diagram dezaktywacji obiektu. Źródło: opracowanie własne. 130
Rysunek 42 Diagram aktywności – anonimizacja klienta AD-KLI-02. Źródło: opracowanie własne. 131
Rysunek 43 Diagram aktywności – przeglądanie historii rezerwacji AD-REZ-07. Źródło: opracowanie własne. 133
Rysunek 44 Diagram aktywności – tworzenie rezerwacji w imieniu klienta AD-REZ-02. Źródło: opracowanie własne. 135
Rysunek 45 Diagram aktywności – wydanie pojazdu AD-OPE-01. Źródło: opracowanie własne. 137
Rysunek 46 Diagram aktywności – odbiór pojazdu AD-OPE-02. Źródło: opracowanie własne. 140
Rysunek 47 Diagram aktywności – zgłoszenie usterki AD-OPE-03. Źródło: opracowanie własne. 142
Rysunek 48 Diagram aktywności – zmiana własnej domeny AD-ZWY-05. Źródło: opracowanie własne. 144
Rysunek 49 Analityczny diagram klas – model całościowy dziedziny (bez atrybutów). Źródło: opracowanie własne. 147
Rysunek 50 Analityczny diagram klas – fragment: perspektywa platformy SaaS. Źródło: opracowanie własne. 148
Rysunek 51 Analityczny diagram klas – fragment: perspektywa floty. Źródło: opracowanie własne. 149
Rysunek 52 Analityczny diagram klas – fragment: perspektywa ofert i rezerwacji. Źródło: opracowanie własne. 150
Rysunek 53 Analityczny diagram klas – fragment: perspektywa finansów i płatności. Źródło: opracowanie własne. 151
Rysunek 54 Diagram architektury systemu. Źródło: opracowanie własne. 155
Rysunek 55 Diagram ilustrujący mechanizm odświeżania tokenu dostępowego. Źródło: opracowanie własne. 158
Rysunek 56 Diagram komponentów ilustrujący przepływ wysyłki wiadomości e-mail Źródło: opracowanie własne. 160
Rysunek 57 Diagram ilustrujący przepływ procesu generowania dokumentu w systemie Wellhicle Źródło: opracowanie własne. 160
Rysunek 58 Wspólne podstawy domeny – klasy bazowe i interfejsy techniczne wykorzystywane przez wszystkie pozostałe diagramy PCD. Źródło: opracowanie własne. 162
Rysunek 59 Katalog obiektów wartości systemu Wellhicle. Źródło: opracowanie własne. 163
Rysunek 60 PCD modułu Tenants – agregat tenanta i powiązane encje konfiguracyjne. Źródło: opracowanie własne. 164
Rysunek 61 PCD modułu Users – hierarchia kont użytkowników z polimorfizmem przez dziedziczenie. Źródło: opracowanie własne. 165
Rysunek 62 PCD modułu Files. Źródło: opracowanie własne. 166
Rysunek 63 PCD modułu ReferenceData – słowniki referencyjne kraju, waluty i języka. Źródło: opracowanie własne. 167
Rysunek 64 PCD modułu Configs – konfiguracja platformy jako dyskryminowane pary klucz-wartość. Źródło: opracowanie własne. 168

Rysunek 65 PCD modułu Translations – polimorficzny mechanizm tłumaczeń wielu typów encji. Źródło: opracowanie własne. 169
Rysunek 66 PCD modułu Vehicles – agregat pojazdu i powiązane encje cyklu użytkowania. Źródło: opracowanie własne. 170
Rysunek 67 PCD modułu Products – interfejs IProduct i klasy implementujące. Źródło: opracowanie własne. 171
Rysunek 68 PCD modułu Offers & Reservations – oferty, rezerwacje i dodatki. Źródło: opracowanie własne. 172
Rysunek 69 PCD modułu Subscriptions – subskrypcje i plany cenowe. Źródło: opracowanie własne. 173
Rysunek 70 PCD modułu Billings – dokumenty rozliczeniowe i ich pozycje. Źródło: opracowanie własne. 174
Rysunek 71 PCD modułu Payments – płatności i metody płatności. Źródło: opracowanie własne. .. 175
Rysunek 72 Enumeracje modułu Vehicles. Źródło: opracowanie własne. 177
Rysunek 73 Enumeracje modułu Reservations. Źródło: opracowanie własne. 177
Rysunek 74 Enumeracje modułu Offers. Źródło: opracowanie własne. 178
Rysunek 75 Enumeracje modułu Payments. Źródło: opracowanie własne. 179
Rysunek 76 Enumeracje modułu Finances (rozliczenia). Źródło: opracowanie własne. 179
Rysunek 77 Enumeracje modułu Subscriptions. Źródło: opracowanie własne. 180
Rysunek 78 Enumeracje modułu Tenants. Źródło: opracowanie własne. 180
Rysunek 79 Enumeracje typów plików i dokumentów. Źródło: opracowanie własne. 181
Rysunek 80 Enumeracje modułu Communication. Źródło: opracowanie własne. 182
Rysunek 81 Wersje logo przygotowane dla systemu Wellhicle. Źródło: opracowanie własne. 183
Rysunek 82 Projekt widoku górnej części strony startowej. Źródło: opracowanie własne. 185
Rysunek 83 Projekt widoku środkowej części strony startowej. Źródło: opracowanie własne. 186
Rysunek 84 Projekt widoku końcowej części strony startowej. Źródło: opracowanie własne. 187
Rysunek 85 Projekt widoku strony rejestracji. Źródło: opracowanie własne. 188
Rysunek 86 Projekt widoku strony logowania. Źródło: opracowanie własne. 189
Rysunek 87 Projekt widoku pierwszego kroku kreatora wypożyczalni. Źródło: opracowanie własne. 190
Rysunek 88 Projekt widoku drugiego kroku kreatora wypożyczalni. Źródło: opracowanie własne. . 191
Rysunek 89 Projekt widoku trzeciego kroku kreatora wypożyczalni. Źródło: opracowanie własne. . 192
Rysunek 90 Projekt widoku listy pojazdów. Źródło: opracowanie własne. 193
Rysunek 91Projekt widoku szczegółów pojazdu. Źródło: opracowanie własne. 194
Rysunek 92 Projekt widoku edycji pojazdu. Źródło: opracowanie własne. 195
Rysunek 93 Projekt widoku dodania pojazdu. Źródło: opracowanie własne. 196
Rysunek 94 Struktura rozwiązania w Visual Studio. Źródło: opracowanie własne. 198
Rysunek 95 Struktura warstw modelu Clean Architecture. Źródło: [103]. 199
Rysunek 96 Struktura projektu Wellhicle.Application w Visual Studio. Źródło: opracowanie własne. 209
Rysunek 97 Okno potwierdzenia akcji usunięcia oddziału. Źródło: opracowanie własne. 244
Rysunek 98 Pole wprowadzania kwoty. Źródło: opracowanie własne. 245
Rysunek 99 Pole wyboru daty i czasu w strefie czasowej oddziału wypożyczalni. Źródło: opracowanie własne. 246
Rysunek 100 Widok sekcji głównej strony publicznej. Źródło: opracowanie własne. 248
Rysunek 101 Widok sekcja informacyjnej strony publicznej. Źródło: opracowanie własne. 248
Rysunek 102 Widok sekcji planów strony publicznej. Źródło: opracowanie własne. 249

Rysunek 103 Stopka strony publicznej. Źródło: opracowanie własne. 249
Rysunek 104 Widok rejestracji właściciela wypożyczalni. Źródło: opracowanie własne. 250
Rysunek 105 Widok logowania pracownika wypożyczalni. Źródło: opracowanie własne. 250
Rysunek 106 Widok kreatora wypożyczalni – krok 1. Źródło: opracowanie własne. 251
Rysunek 107 Widok kreatora wypożyczalni – krok 2. Źródło: opracowanie własne. 252
Rysunek 108 Widok kreatora wypożyczalni – krok 3. Źródło: opracowanie własne. 252
Rysunek 109 Widok kreatora wypożyczalni – krok 4. Źródło: opracowanie własne. 254
Rysunek 110 Widok kreatora wypożyczalni – krok 5. Źródło: opracowanie własne. 254
Rysunek 111 Widok nawigacji. Źródło: opracowanie własne. 255
Rysunek 112 Widok paska narzędzi. Źródło: opracowanie własne. 256
Rysunek 113 Widok listy pojazdów. Źródło: opracowanie własne. 258
Rysunek 114 Widok filtrów listy pojazdów. Źródło: opracowanie własne. 258
Rysunek 115 Okno listy pojazdów w trybie wyboru. Źródło: opracowanie własne. 262
Rysunek 116 Okno edycji pojazdu. Źródło: opracowanie własne. 263
Rysunek 117 Widok komponentu kadrowania zdjęć w formularzu dodawania/edycji pojazdu. Źródło: opracowanie własne. 264
Rysunek 118 Okno szczegółów oddziału. Źródło: opracowanie własne. 265
Rysunek 119 Okno formularza tworzenia szablonu wiadomości. Źródło: opracowanie własne. 266
Rysunek 120 Sekcja wyglądu strony publicznej. Źródło: opracowanie własne. 267
Rysunek 121 Sekcja plików strony publicznej. Źródło: opracowanie własne. 267
Rysunek 122 Widok aktualnej subskrypcji. Źródło: opracowanie własne. 268
Rysunek 123 Widok dostępnych planów subskrypcji. Źródło: opracowanie własne. 268
Rysunek 124 Widok kreatora zmiany planu subskrypcji – krok 1. Źródło: opracowanie własne. 269
Rysunek 125 Widok podglądu rozliczenia zmiany subskrypcji. Źródło: opracowanie własne. 269
Rysunek 126 Widok historii subskrypcji. Źródło: opracowanie własne. 270
Rysunek 127 Sekcja ustawień ogólnych. Źródło: opracowanie własne. 271
Rysunek 128 Sekcja ustawień konta SMTP. Źródło: opracowanie własne. 271
Rysunek 129 Sekcja ustawień szablonów. Źródło: opracowanie własne. 272
Rysunek 130 Kreator podłączania domeny - krok 1. Źródło: opracowanie własne. 273
Rysunek 131 Kreator podłączania domeny - krok 2. Źródło: opracowanie własne. 273
Rysunek 132 Kreator podłączania domeny - krok 3. Źródło: opracowanie własne. 274
Rysunek 133 Kreator podłączania domeny - krok 4. Źródło: opracowanie własne. 274
Rysunek 134 Sekcja ustawień domeny. Źródło: opracowanie własne. 275
Rysunek 135. Widok aktywacji wypożyczalni. Źródło: opracowanie własne. 275
Rysunek 136 Widok odnowienia subskrypcji. Źródło: opracowanie własne. 276
Rysunek 137 Widok kreatora subskrypcji - krok 1. Źródło: opracowanie własne. 276
Rysunek 138 Widok kreatora subskrypcji - krok 2. Źródło: opracowanie własne. 277
Rysunek 139 Widok kreatora subskrypcji - krok 3. Źródło: opracowanie własne. 277
Rysunek 140 Widok kreatora subskrypcji - krok 4. Źródło: opracowanie własne. 278
Rysunek 141 Przykładowy widok aplikacji do administrowania wypożyczalniami. Źródło: opracowanie własne. 278
Rysunek 142 Widok listy ofert uruchomionej pod subdomeną najemcy. Źródło: opracowanie własne. 281
Rysunek 143 Widok strony głównej aplikacji w kolorystyce czerwonej w motywie jasnym. Źródło: opracowanie własne. 282
Rysunek 144 Widok strony głównej aplikacji w kolorystyce niebieskiej w motywie ciemnym. Źródło: opracowanie własne. 283

Rysunek 145 Widok strony głównej po zmianie kroju pisma na Comic Sans. Źródło: opracowanie własne. 284
Rysunek 146 Widok listy ofert z filtrami. Źródło: opracowanie własne. 285
Rysunek 147 Widok szczegółów oferty z przyciskiem rozpoczęcia rezerwacji. Źródło: opracowanie własne. 285
Rysunek 148 Fragment formularza rezerwacji. Źródło: opracowanie własne. 286
Rysunek 149 Widok podsumowania rezerwacji. Źródło: opracowanie własne. 287
Rysunek 150 Widok formularza danych osobowych w panelu klienta. Źródło: opracowanie własne. 288
Rysunek 151 Widok rozwiniętych szczegółów rezerwacji. Źródło: opracowanie własne. 289
Rysunek 152 Ekran logowania aplikacji Wellhicle. Źródło: opracowanie własne. 294
Rysunek 153. Dialog informujący o braku podpisu pracownika z opcją przejścia do ekranu profilu. Źródło: opracowanie własne. 299
Rysunek 154. Ekran zgłaszania usterki pojazdu. Źródło: opracowanie własne. 300
Rysunek 155 Widok panelu ze śladami aplikacyjnymi oraz metrykami z Grafany. Źródło: opracowanie własne. 307
Rysunek 156 Uproszczony przepływ wdrażania aplikacji do zarządzania wypożyczalnią. Źródło: opracowanie własne. 311
Rysunek 157 Przykładowy widok zgłoszonego błędu. Źródło: opracowanie własne. 319
Rysunek 158 Konfiguracja zestawu zmiennych środowiskowych w narzędziu Postman. Źródło: opracowanie własne. 320
Rysunek 159 Wykorzystanie zmiennych środowiskowych w ciele żądania. Źródło: opracowanie własne. 320
Rysunek 160 Przykładowe użycie Scalara w celu przetestowania API. Źródło: opracowanie własne. 321
Rysunek 161 Przykład błędu wykrytego podczas testów manualnych. Źródło: opracowanie własne. 323
Rysunek 162 Raport wykonanych testów w narzędziu Playwright. Źródło: opracowanie własne. 327
Rysunek 163 Widok zrealizowanego testu w raporcie Playwright. Źródło: opracowanie własne. 328

12.5. Spis listingów

Listing 1 Fragment konfiguracji osadzający autorski skrypt do narzędzia Scalar. Źródło: opracowanie własne. 57
Listing 2 Klasa `AggregateRoot` z projektu `Wellhicle.Domain`. Źródło: opracowanie własne. 201
Listing 3 Klasa `Email` z projektu `Wellhicle.Domain`. Źródło: opracowanie własne. 201
Listing 4 Fragment klasy `Reservation` z projektu `Wellhicle.Domain` - metoda `UpdateStatus`. Źródło: opracowanie własne. 204
Listing 5 Fragment klasy `User` z projektu `Wellhicle.Domain` – metoda `ChangePassword`. Źródło: opracowanie własne. 205
Listing 6 Interfejsy `ICommandHandler` oraz `IQueryHandler` z projektu `Wellhicle.Application`. Źródło: opracowanie własne. 206
Listing 7 Fragment metody `HandleAsync` z klasy `CreateReservationCommandHandler`, z projektu `Wellhicle.Application`. Źródło: opracowanie własne. 207

Listing 8 Klasa `Result` z projektu `Wellhicle.Application`. Źródło: opracowanie własne. 209
Listing 9 Interfejs `IUserContext` z projektu `Wellhicle.Application`. Źródło: opracowanie własne. 211
Listing 10 Fragment implementacji klasy `UnitOfWork` z projektu `Wellhicle.Infrastructure`. Źródło: Opracowanie własne. 212
Listing 11 Metoda `PublishDomainEventsAsync` z klasy `UnitOfWork`, z projektu `Wellhicle.Infrastructure`. Źródło: opracowanie własne. 214
Listing 12 Metoda `ProjectWithStatus` z klasy `VehicleAccessReadRepository`, z projektu `Wellhicle.Infrastructure`. Źródło: opracowanie własne. 215
Listing 13 Metoda `IsUserAllowedForFrontend` z klasy `LoginCommand`, z projektu `Wellhicle.Application`. Źródło: opracowanie własne. 216
Listing 14 Metoda `CreateErrorResult` z klasy `ResultExtension`, z projektu `Wellhicle.Presentation`. Źródło: opracowanie własne. 219
Listing 15 Fragment klasy `CorrelationIdMiddleware` z projektu `Wellhicle.Presentation`. Źródło: opracowanie własne. 221
Listing 16 Fragment klasy `RecurringJobsRegistration` z projektu `Wellhicle.Presentation`. Źródło: opracowanie własne. 222
Listing 17 Wybrane metody z interfejsu IDocumentService. Źródło: opracowanie własne. 224
Listing 18 Fragment pliku appsettings.json przedstawiający konfigurację zbiorów i folderów w magazynie RustFS. Źródło: opracowanie własne. 225
Listing 19 Fragment klasy RustFileStorageService odpowiedzialny za generowanie adresów URL obiektów. Źródło: opracowanie własne. 226
Listing 20 Fragment konfiguracji zapisu do tabeli Files wykorzystujący TPH. Źródło: opracowanie własne. 227
Listing 21 Metoda sprawdzająca poprawność przesłanego pliku. Źródło: opracowanie własne. 227
Listing 22 Reguły walidacji plików. Źródło: opracowanie własne. 229
Listing 23 Mechanizm usuwania pliku po nieudanym zapisie metadanych. Źródło: opracowanie własne. 229
Listing 24 Fragment interceptora odpowiedzialny za kasowanie plików z magazynu. Źródło: opracowanie własne. 230
Listing 25 Fragment metody `GenerateDocument()` odpowiedzialnej za główny przepływ generatora. Źródło: opracowanie własne. 231
Listing 26 Metoda `Resolve()`, służąca do zarządzania przepływem rozwiązywania pól na szablonie. Źródło: opracowanie własne. 233
Listing 27 Metoda `ResolvePath()` odczytująca rzeczywiste wartości z właściwości obiektów. Źródło: opracowanie własne. 234
Listing 28 Metoda `TryExtractMergeKey()` służąca do znajdowania pól tymczasowych w dokumencie. Źródło: opracowanie własne. 234
Listing 29 Metoda `_expandCollectionTemplateRows()` odpowiedzialna za stworzenie wiersza tabeli dla każdego elementu kolekcji. Źródło: opracowanie własne. 235
Listing 30 Klasa `PdfGenerator` wraz z metodą `GeneratePdfAsync()`, odpowiedzialną za komunikację z FastAPI. Źródło: opracowanie własne. 236
Listing 31 Fragment punktu końcowego FastAPI odpowiedzialnego za konwertowanie DOCX na PDF. Źródło: opracowanie własne. 236
Listing 32 Metoda `_tryHandleSignatureMergeField()` ukrywająca znaczniki podpisów. Źródło: opracowanie własne. 237

Listing 33 Fragment metody `AddDomainAsync` z klasy `DomainConfigurationService`, z projektu `Wellhicle.Application`. Źródło: opracowanie własne. 239
Listing 34 Funkcja przechwytująca żądania. Źródło: opracowanie własne. 243
Listing 35 Metoda obsługująca żądanie odświeżenia sesji. Źródło: opracowanie własne. 243
Listing 36 Konfiguracja okna potwierdzenia akcji. Źródło: opracowanie własne. 244
Listing 37 Metoda wyliczająca kwotę dla najmniejszej możliwej jednostki waluty. Źródło: opracowanie własne. 245
Listing 38 Przekształcanie tekstu wprowadzonego przez użytkownika na obiekt daty. Źródło: opracowanie własne. 246
Listing 39 Funkcja asynchronicznie walidująca subdomenę. Źródło: opracowanie własne. 253
Listing 40 Inicjalizacja strumieni oraz metody odczytu i zapisu oddziałów. Źródło: opracowanie własne. 256
Listing 41 Propagacja zmiany do BehaviorSubject po operacji zapisu oddziału. Źródło: opracowanie własne. 257
Listing 42 Fragmenty kodu szablonu komponentu wyboru oddziałów. Źródło: opracowanie własne. 257
Listing 43 Budowanie obiektu żądania pobrania danych do listy pojazdów. Źródło: opracowanie własne. 259
Listing 44 Fragmenty kodu szablonu listy pojazdów. Źródło: opracowanie własne. 259
Listing 45. Interfejs listy w trybie wyboru. Źródło: opracowanie własne. 260
Listing 46 Fragmenty kodu komponentu listy pojazdów. Źródło: opracowanie własne 260
Listing 47 Klasa MultipleSelection rozszerzająca SelectionModel. Źródło: opracowanie własne. 260
Listing 48 Fragmenty kodu szablonu komponentu listy pojazdów. Źródło: opracowanie własne. 261
Listing 49 Pole wyboru pojazdu w trybie listy. Źródło: opracowanie własne. 261
Listing 50 Pomocnicza funkcja formRecord tworząca grupę kontrolek opisu dla obsługiwanych przez wypożyczalnię języków. Źródło: opracowanie własne. 263
Listing 51 Metoda otwierająca aplikację wypożyczalni i przekazująca dane uwierzytelniające administratora. Źródło: opracowanie własne. 279
Listing 52 Fragment komponentu odbierającego dane uwierzytelniające w aplikacji do zarządzania wypożyczalnią. Źródło: opracowanie własne. 279
Listing 53 Wyznaczanie subdomeny najemcy. Źródło: opracowanie własne. 280
Listing 54 Nawigacja najwyższego poziomu reagująca na zmianę stanu sesji – wylogowanie czyści cały stos nawigacyjny. Źródło: opracowanie własne. 289
Listing 55 Bazowa wtyczka konwencji Gradle konfigurująca cele kompilacji KMP dla Androida i iOS. Źródło: opracowanie własne. 292
Listing 56 Wtyczka konwencji Gradle dodająca wspólne zależności do wszystkich modułów feature. Źródło: opracowanie własne. 292
Listing 57 Odświeżanie tokenu sesji przy każdym starcie aplikacji – sukces pomija ekran logowania. Źródło: opracowanie własne. 293
Listing 58 Harmonogram automatycznego odświeżania tokenu z minutowym wyprzedzeniem przed jego wygaśnięciem. Źródło: opracowanie własne. 294
Listing 59 Niemutowalna klasa stanu procesu wydania pojazdu obejmująca wszystkie etapy wielokrokowego wizarda. Źródło: opracowanie własne. 295
Listing 60 Zdarzenia nawigacyjne procesu wydania pojazdu emitowane jako SharedFlow dla jednokrotnego przetworzenia. Źródło: opracowanie własne. 295
Listing 61 Pamięć podręczna URL kontraktu PDF indeksowana po identyfikatorze rezerwacji. Źródło: opracowanie własne. 296

Listing 62 Wybór punktu wejścia do procesu wydania na podstawie statusu rezerwacji – wznowienie pomija weryfikację klienta. Źródło: opracowanie własne. 296
Listing 63 Niemutowalna klasa stanu procesu zwrotu pojazdu obejmująca wszystkie etapy wielokrokowego wizarda. Źródło: opracowanie własne. 297
Listing 64 Zdarzenia nawigacyjne procesu zwrotu pojazdu emitowane jako SharedFlow dla jednokrotnego przetworzenia. Źródło: opracowanie własne. 298
Listing 65 Walidacja formularza inspekcji zwrotu – wymaga zdjęć przy stwierdzonych nieprawidłowościach stanu pojazdu. Źródło: opracowanie własne. 298
Listing 66 Pamięć podręczna URL protokołu zwrotu PDF indeksowana po identyfikatorze zwrotu. Źródło: opracowanie własne. 298
Listing 67 Interfejs komponentu aparatu/galerii z asynchronicznymi metodami zwracającymi null przy anulowaniu. Źródło: opracowanie własne. 301
Listing 68 Implementacja aparatu na Androidzie zawieszająca coroutine do powrotu wyniku z ActivityResultLauncher . Źródło: opracowanie własne. 301
Listing 69 Deklaracja efektu blokady orientacji poziomej z pustą implementacją po stronie iOS. Źródło: opracowanie własne. 302
Listing 70 Blokada orientacji poziomej ekranu na czas zbierania podpisu z przywróceniem oryginału po opuszczeniu ekranu. Źródło: opracowanie własne. 302
Listing 71 Nadpisanie konfiguracji lokalizacji dla drzewa composable bez restartu aktywności na Androidzie. Źródło: opracowanie własne. 302
Listing 72 Pusta implementacja zmiany języka przez composable na iOS – deleguje do NSUserDefaults w repozytorium. Źródło: opracowanie własne. 303
Listing 73 Ustawienie preferowanego języka aplikacji w NSUserDefaults na iOS. Źródło: opracowanie własne. 303
Listing 74 Deklaracja expect komponentu wyświetlającego dokumenty PDF wymagająca implementacji per platforma. Źródło: opracowanie własne. 303
Listing 75 Renderowanie stron PDF do Bitmap z lokalnym buforowaniem pliku na Androidzie. Źródło: opracowanie własne. 303
Listing 76 Wyświetlanie dokumentu PDF przez WKWebView osadzone w composable za pomocą UIKitView na iOS. Źródło: opracowanie własne. 304
Listing 77 Wybór punktu wejścia przy wznawianiu zwrotu – pomija inspekcję gdy protokół jest dostępny w pamięci podręcznej. Źródło: opracowanie własne. 305
Listing 78 Usunięcie URL protokołu z pamięci podręcznej po zakończeniu procesu zwrotu. Źródło: opracowanie własne. 305
Listing 79 Fragment konfiguracji konta serwisowego używanego do wdrażania aplikacji w klastrze Kubernetes. Źródło: opracowanie własne. 307
Listing 80 Fragment przepływu dla aplikacji do korzystania z usług wypożyczalni Źródło: opracowanie własne. 309
Listing 81 Treść pliku Dockerfile dla aplikacji frontendowych Źródło: opracowanie własne. 311
Listing 82 Fragment przepływu wdrożeniowego uruchamiający migracje bazy danych. Źródło: opracowanie własne. 312
Listing 83 Interfejs konektora platformy Wellhicle – operacje wymiany danych. Źródło: opracowanie własne. 314
Listing 84 Pobranie strony ofert przez typowaną trasę wygenerowanego klienta – bez ręcznego budowania adresu i serializacji, z ujednoliconą obsługą błędów. Źródło: opracowanie własne. 314
Listing 85 Deklaracja rozszerzenia encji towaru w pliku business.xml – tabela TowarWellhicle dowiązana do encji Towar. Źródło: opracowanie własne. 315

Listing 86 Fragment klasy `AdditionPage` implementująca wzorzec POM. Źródło: opracowanie własne. 324
Listing 87 Fragment pliku konfiguracyjnego Playwright z uwzględnieniem zależności pomiędzy zestawami testów. Źródło: opracowanie własne. 325
Listing 88 Implementacja ścieżki głównej testów dodatków. Źródło: opracowanie własne. 325
Listing 89 Przykładowy test jednostkowy realizujący wzorzec AAA. Źródło: Opracowanie własne. 329
Listing 90 Przykładowy test jednostkowy warstwy infrastruktury realizujący wzorzec AAA. Źródło: opracowanie własne. 330
Listing 91 Przykładowy test jednostkowy warstwy prezentacji z atrybutem Theory. Źródło: opracowanie własne. 331

12.6. Cytowane prace

[1] Rentsyst Ltd., „RentSyst,” 06 2026. [Online]. Available: https://rentsyst.com/pl/about/. [Data uzyskania dostępu: 29 maja 2026].

[2] HQ Rental Software, „HQ Rental Software,” 06 2026. [Online]. Available: https://hqrentalsoftware.com/. [Data uzyskania dostępu: 27 maja 2026].

[3] Renthelp Sp. z o.o., „Renthelp,” 06 2026. [Online]. Available: https://renthelp.io/pl/. [Data uzyskania dostępu: 26 kwietnia 2026].

[4] FleetWire, „FleetWire,” 06 2026. [Online]. Available: https://fleetwire.io/. [Data uzyskania dostępu: 11 maja 2026].

[5] Atlassian, „Jira,” Atlassian, 05 2026. [Online]. Available: https://www.atlassian.com/software/jira/features. [Data uzyskania dostępu: 2 czerwca 2026].

[6] Atlassian, „Confluence,” Attlasian, 05 2026. [Online]. Available: https://www.atlassian.com/software/confluence/features. [Data uzyskania dostępu: 26 kwietnia 2026].

[7] GitHub, „GitHub Docs,” GitHub, 05 2026. [Online]. Available: https://docs.github.com/en. [Data uzyskania dostępu: 28 maja 2026].

[8] Microsoft, „Microsoft Teams,” Microsoft, 05 2026. [Online]. Available: https://www.microsoft.com/pl-pl/microsoft-teams. [Data uzyskania dostępu: 4 maja 2026].

[9] Discord, „Discord,” Discord, 05 2026. [Online]. Available: https://discord.com/. [Data uzyskania dostępu: 23 kwietnia 2026].

[10] Microsoft, „Microsoft OneDrive,” Microsoft, 05 2026. [Online]. Available: https://www.microsoft.com/pl-pl/microsoft-365/onedrive/online-cloud-storage. [Data uzyskania dostępu: 30 kwietnia 2026].

[11] D. Rapptz, „discord.py,” [Online]. Available: https://discordpy.readthedocs.io. [Data uzyskania dostępu: 6 maja 2026].

[12] Lucid Software, „Diagramming powered by intelligence,” [Online]. Available: https://www.lucidchart.com/pages. [Data uzyskania dostępu: 18 maja 2026].

[13] draw.io Ltd, „Security-first diagramming for teams,” [Online]. Available: https://www.drawio.com/. [Data uzyskania dostępu: 12 maja 2026].

[14] Microsoft, „Microsoft Visio: tworzenie diagramów i schematy | Microsoft 365,” [Online]. Available: https://www.microsoft.com/pl-pl/microsoft-365/visio. [Data uzyskania dostępu: 18 maja 2026].

[15] Sparx Systems, 07 06 2026. [Online]. Available: https://sparxsystems.com/products/ea/. [Data uzyskania dostępu: 14 maja 2026].

[16] Figma, „Figma Design / Free Design Tool for Websites, Product Design & More,” Figma, 6 Maj 2026. [Online]. Available: https://www.figma.com/design/. [Data uzyskania dostępu: 18 maja 2026].

[17] Figma, „Figma: The Collaborative Interface Design Tool,” [Online]. Available: https://www.figma.com. [Data uzyskania dostępu: 8 maja 2026].

[18] Inkscape, Współtwórcy projektu, „Free and Open Source Vector Drawing,” 3 Marzec 2022. [Online]. Available: https://github.com/inkscape/inkscape. [Data uzyskania dostępu: 14 maja 2026].

[19] Google, „Material Design 3,” 6 Maj 2026. [Online]. Available: https://m3.material.io/. [Data uzyskania dostępu: 6 czerwca 2026].

[20] Y. Evjen, „Introducing Material Theme Builder,” 27 Październik 2021. [Online]. Available: https://m3.material.io/blog/material-theme-builder. [Data uzyskania dostępu: 12 maja 2026].

[21] Microsoft, „Microsoft, Visual Studio IDE - Documentation,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/visualstudio/ide/. [Data uzyskania dostępu: 20 maja 2026].

[22] JetBrains, „JetBrains, Rider - The Cross-Platform .NET IDE,” JetBrains, 05 2026. [Online]. Available: https://www.jetbrains.com/rider/. [Data uzyskania dostępu: 26 maja 2026].

[23] S. Chacon, „Git - Distributed Version Control System,” 05 2026. [Online]. Available: https://git-scm.com/. [Data uzyskania dostępu: 30 maja 2026].

[24] Microsoft, „Microsoft, SQL Server Management Studio (SSMS),” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/sql/ssms/sql-server-management-studio-ssms. [Data uzyskania dostępu: 31 maja 2026].

[25] Docker Inc., „Docker - Documentation,” Docker Inc., 05 2026. [Online]. Available: https://docs.docker.com/. [Data uzyskania dostępu: 22 maja 2026].

[26] Docker Inc., „Docker Compose Overview,” Docker Inc., 05 2026. [Online]. Available: https://docs.docker.com/compose/. [Data uzyskania dostępu: 27 kwietnia 2026].

[27] Microsoft, „Microsoft, C# documentation,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/dotnet/csharp/. [Data uzyskania dostępu: 23 maja 2026].

[28] Microsoft, „Microsoft, What's new in .NET 8,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/dotnet/core/whats-new/dotnet-8. [Data uzyskania dostępu: 18 czerwca 2026].

[29] Microsoft, „Microsoft, Microsoft SQL Server - Documentation,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/sql/sql-server/. [Data uzyskania dostępu: 8 maja 2026].

[30] Microsoft, „Microsoft, Entity Framework Core,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/ef/core/. [Data uzyskania dostępu: 20 kwietnia 2026].

[31] Microsoft, „Microsoft, ASP.NET Core documentation,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/. [Data uzyskania dostępu: 4 czerwca 2026].

[32] FluentValidation Project, „FluentValidation - Documentation,” FluentValidation Project, 05 2026. [Online]. Available: https://docs.fluentvalidation.net/. [Data uzyskania dostępu: 29 maja 2026].

[33] OpenAPI Initiative, „OpenAPI Specification,” OpenAPI Initiative, 05 2026. [Online]. Available: https://spec.openapis.org/oas/latest.html. [Data uzyskania dostępu: 11 maja 2026].

[34] Amazon, „Amazon Web Services, AWS SDK for .NET - Amazon Simple Storage Service,” Amazon, 05 2026. [Online]. Available: https://docs.aws.amazon.com/sdk-for-net/v3/developer-guide/csharp_s3_code_examples.html. [Data uzyskania dostępu: 27 maja 2026].

[35] Six Labors, „Six Labors, ImageSharp - Cross-platform .NET image processing,” Six Labors, 05 2026. [Online]. Available: https://docs.sixlabors.com/articles/imagesharp/. [Data uzyskania dostępu: 6 maja 2026].

[36] Microsoft, „Microsoft, Welcome to the Open XML SDK 3.0 for Office,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/office/open-xml/open-xml-sdk. [Data uzyskania dostępu: 15 czerwca 2026].

[37] iText Group, „iText Core - Open Source PDF library for .NET.,” iText Group, 05 2026. [Online]. Available: https://itextpdf.com/products/itext-core. [Data uzyskania dostępu: 15 czerwca 2026].

[38] RabbitMQ Team, „RabbitMQ - One broker to queue them all,” RabbitMQ Team, 05 2026. [Online]. Available: https://www.rabbitmq.com/. [Data uzyskania dostępu: 17 czerwca 2026].

[39] Hangfire OÜ, „Hangfire - Background processing for .NET,” Hangfire OÜ, 05 2026. [Online]. Available: https://www.hangfire.io/. [Data uzyskania dostępu: 3 maja 2026].

[40] Microsoft, „Microsoft, Cache in-memory in ASP.NET Core,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/performance/caching/memory. [Data uzyskania dostępu: 25 kwietnia 2026].

[41] M. Jones, J. Bradley i N. Sakimura, „JSON Web Token (JWT), RFC 7519, Internet Engineering Task Force (IETF),” maj 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7519.html. [Data uzyskania dostępu: 16 kwietnia 2026].

[42] Microsoft, „Microsoft, PasswordHasher<TUser> Class - Microsoft.AspNetCore.Identity,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/dotnet/api/microsoft.aspnetcore.identity.passwordhasher-1. [Data uzyskania dostępu: 30 kwietnia 2026].

[43] N. Blumhardt, „Serilog - Structured logging for .NET,” Serilog Team, 05 2026. [Online]. Available: https://serilog.net/. [Data uzyskania dostępu: 28 maja 2026].

[44] Microsoft, „Microsoft, .NET Generic Host,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/dotnet/core/extensions/generic-host. [Data uzyskania dostępu: 12 czerwca 2026].

[45] Microsoft, „Microsoft, Configuration in ASP.NET Core,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/fundamentals/configuration/. [Data uzyskania dostępu: 16 czerwca 2026].

[46] Microsoft, „Microsoft, Health checks in ASP.NET Core,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/aspnet/core/host-and-deploy/health-checks. [Data uzyskania dostępu: 16 kwietnia 2026].

[47] Polly Contributors, „Polly - Resilience and transient-fault-handling library for .NET,” Polly Contributors, 05 2026. [Online]. Available: https://www.pollydocs.org/. [Data uzyskania dostępu: 12 maja 2026].

[48] T. i. w. Clegg, „libphonenumber-csharp - C# port of Google's libphonenumber library,” 05 2026. [Online]. Available: https://github.com/twcclegg/libphonenumber-csharp. [Data uzyskania dostępu: 25 maja 2026].

[49] Google, „libphonenumber — Google's common Java, C++ and JavaScript library for parsing, formatting, and validating international phone numbers,” Google, 05 2026. [Online]. Available: https://github.com/google/libphonenumber. [Data uzyskania dostępu: 29 maja 2026].

[50] [Online]. Available: https://angular.dev/overview. [Data uzyskania dostępu: 14 kwietnia 2026].

[51] [Online]. Available: https://survey.stackoverflow.co/2025/technology. [Data uzyskania dostępu: 4 maja 2026].

[52] [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html#typescript-a-static-type-checker. [Data uzyskania dostępu: 15 maja 2026].

[53] [Online]. Available: https://rxjs.dev/guide/overview. [Data uzyskania dostępu: 12 czerwca 2026].

[54] [Online]. Available: https://v20.material.angular.dev. [Data uzyskania dostępu: 26 kwietnia 2026].

[55] [Online]. Available: https://v20.material.angular.dev/guide/theming. [Data uzyskania dostępu: 24 kwietnia 2026].

[56] [Online]. Available: https://eslint.org/docs/latest/about/. [Data uzyskania dostępu: 25 kwietnia 2026].

[57] [Online]. Available: https://prettier.io/docs/. [Data uzyskania dostępu: 4 czerwca 2026].

[58] [Online]. Available: https://ngx-translate.org/. [Data uzyskania dostępu: 16 kwietnia 2026].

[59] [Online]. Available: https://angular.dev/guide/i18n. [Data uzyskania dostępu: 26 maja 2026].

[60] JetBrains, „What is Kotlin Multiplatform,” 18 Listopad 2025. [Online]. Available: https://kotlinlang.org/docs/multiplatform/kmp-overview.html. [Data uzyskania dostępu: 2026 06 19].

[61] JetBrains, „Compose Multiplatform,” 2025. [Online]. Available: https://www.jetbrains.com/compose-multiplatform/. [Data uzyskania dostępu: 30 maja 2026].

[62] JetBrains, „Get started with Compose Multiplatform,” 2025. [Online]. Available: https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-multiplatform-getting-started.html . [Data uzyskania dostępu: 22 kwietnia 2026].

[63] JetBrains, „Ktor - Create asynchronous client and server applications,” 2025. [Online]. Available: https://ktor.io/. [Data uzyskania dostępu: 14 kwietnia 2026].

[64] JetBrains, „Ktor HTTP Client Engines,” 2025. [Online]. Available: https://ktor.io/docs/welcome.html. [Data uzyskania dostępu: 5 maja 2026].

[65] JetBrains, „kotlinx.serialization,” 2025. [Online]. Available: https://kotlinlang.org/docs/serialization.html. [Data uzyskania dostępu: 16 czerwca 2026].

[66] Koin Team, „Koin - The pragmatic Kotlin & Kotlin Multiplatform Dependency Injection framework,” 2025. [Online]. Available: https://insert-koin.io/ . [Data uzyskania dostępu: 12 maja 2026].

[67] Koin Team, „Koin for Kotlin Multiplatform,” 2025. [Online]. Available: https://insert-koin.io/docs/reference/koin-mp/kmp/. [Data uzyskania dostępu: 6 maja 2026].

[68] W. Russell, „multiplatform-settings,” 2025. [Online]. Available: https://github.com/russhwolf/multiplatform-settings . [Data uzyskania dostępu: 25 maja 2026].

[69] O. Kazuhiro, „BuildKonfig - BuildConfig for Kotlin Multiplatform Project,” 2025. [Online]. Available: https://github.com/yshrsmz/BuildKonfig. [Data uzyskania dostępu: 15 czerwca 2026].

[70] Google, „Firebase App Distribution,” 2025. [Online]. Available: https://firebase.google.com/docs/app-distribution. [Data uzyskania dostępu: 12 maja 2026].

[71] Google, „Navigation with Compose,” 06 2026. [Online]. Available: https://developer.android.com/develop/ui/compose/navigation. [Data uzyskania dostępu: 11 maja 2026].

[72] JetBrains, „Compose Multiplatform,” 06 2026. [Online]. Available: https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-navigation-routing.html. [Data uzyskania dostępu: 10 czerwca 2026].

[73] OpenTelemetry, [Online]. Available: https://opentelemetry.io/. [Data uzyskania dostępu: 20 maja 2026].

[74] Nginx, [Online]. Available: https://nginx.org/. [Data uzyskania dostępu: 10 maja 2026].

[75] Kubernetes, [Online]. Available: https://kubernetes.io/. [Data uzyskania dostępu: 19 maja 2026].

[76] Cloudflare, [Online]. Available: https://developers.cloudflare.com/tunnel/. [Data uzyskania dostępu: 20 maja 2026].

[77] Harbor, [Online]. Available: https://goharbor.io/. [Data uzyskania dostępu: 21 maja 2026].

[78] RustFS, [Online]. Available: https://rustfs.com/. [Data uzyskania dostępu: 21 maja 2026].

[79] Prometheus, [Online]. Available: https://prometheus.io/. [Data uzyskania dostępu: 18 maja 2026].

[80] Grafana Labs, [Online]. Available: https://grafana.com/oss/loki/. [Data uzyskania dostępu: 19 maja 2026].

[81] Grafana Labs, [Online]. Available: https://grafana.com/oss/tempo/. [Data uzyskania dostępu: 20 maja 2026].

[82] Grafana Labs, [Online]. Available: https://grafana.com/. [Data uzyskania dostępu: 20 maja 2026].

[83] The Linux Foundation, „k3s," [Online]. Available: https://k3s.io/. [Data uzyskania dostępu: 26 maja 2026].

[84] The Linux Foundation, „Helm," [Online]. Available: https://helm.sh/. [Data uzyskania dostępu: 25 maja 2026].

[85] Github, „Github Actions," [Online]. Available: https://github.com/features/actions. [Data uzyskania dostępu: 21 maja 2026].

[86] „Kustomize," [Online]. Available: https://kustomize.io/. [Data uzyskania dostępu: 22 maja 2026].

[87] J. Świątkowski, „Co to jest Postman, do czego służy i jakie są jego funkcjonalności?," 24 06 2022. [Online]. Available: https://www.droptica.pl/blog/co-jest-postman-do-czego-sluzy-i-jakie-sa-jego-funkcjonalnosci/. [Data uzyskania dostępu: 02 maja 2026].

[88] A. Michalski, „Insomnia: Kompletna platforma do projektowania, testowania i mockowania API #EN331," 19 10 2025. [Online]. Available: https://uxairforce.pl/insomnia-kompletna-platforma-do-projektowania-testowania-i-mockowania-api-en331. [Data uzyskania dostępu: 02 maja 2026].

[89] P. Bykowski, „Swagger UI – przejrzysta wizualizacja zasobów API," 12 5 2020. [Online]. Available: https://bykowski.pl/swagger-ui-przejrzysta-wizualizacja-zasobow-api/. [Data uzyskania dostępu: 04 maja 2026].

[90] V. Bayrak, „Scalar for .NET Minimal APIs: A Modern Alternative to Swagger UI," 15 02 2026. [Online]. Available: https://medium.com/@wbayrakvlad/scalar-for-net-minimal-apis-a-modern-alternative-to-swagger-ui-eca59d71f611. [Data uzyskania dostępu: 04 maja 2026].

[91] Scalar, „Scalar - API References & Documentation," Scalar, 05 2026. [Online]. Available: https://scalar.com/. [Data uzyskania dostępu: 24 maja 2026].

[92] Testerzy.pl, „Wstęp do Playwright," 23 09 2024. [Online]. Available: https://testerzy.pl/baza-wiedzy/narzedzia/wstep-do-playwright. [Data uzyskania dostępu: 04 maja 2026].

[93] .NET Foundation, „About xUnit.net," [Online]. Available: https://xunit.net/?tabs=cs. [Data uzyskania dostępu: 04 maja 2026].

[94] K. Fakhroutdinov, „Actions," 2026. [Online]. Available: https://www.uml-diagrams.org/activity-diagrams-actions.html. [Data uzyskania dostępu: 14 maja 2026].

[95] Microsoft Learn, „Multitenant SaaS Patterns - Azure SQL Database," [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-sql/database/saas-tenancy-app-design-patterns. [Data uzyskania dostępu: 6 maja 2026].

[96] „MDN Web Docs (Mozilla)," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Defenses/Same-origin_policy. [Data uzyskania dostępu: 5 maja 2026].

[97] „Cross Site Request Forgery (CSRF)," [Online]. Available: https://owasp.org/www-community/attacks/csrf. [Data uzyskania dostępu: 3 maja 2026].

[98] T. Haerder i A. Reuter, „Principles of Transaction-Oriented Database Recovery," ACM Computing Surveys, 1983.

[99] Hibernate ORM, „What is an ORM?," [Online]. Available: https://hibernate.org/orm/what-is-an-orm/. [Data uzyskania dostępu: 6 maja 2026].

[100] Microsoft Learn, „Buforowanie połączeń SQL Server," [Online]. Available: https://learn.microsoft.com/pl-pl/dotnet/framework/data/adonet/sql-server-connection-pooling. [Data uzyskania dostępu: 4 maja 2026].

[101] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, Tomy %1 z %2978-0-13-449416-4, Prentice Hall, 2017.

[102] Microsoft, „Common web application architectures, Microsoft Learn," Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures#clean-architecture. [Data uzyskania dostępu: 8 maja 2026].

[103] NDepend, „NDepend Blog," 06 2026. [Online]. Available: https://blog.ndepend.com/clean-architecture-for-asp-net-core-solution/. [Data uzyskania dostępu: 27 kwietnia 2026].

[104] R. C. Martin, „The Clean Architecture, The Clean Code Blog," 05 2026. [Online]. Available: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. [Data uzyskania dostępu: 10 czerwca 2026].

[105] Microsoft, „Designing a DDD-oriented microservice, Microsoft Learn," Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice. [Data uzyskania dostępu: 25 maja 2026].

[106] Microsoft, „Architect multitenant solutions on Azure, Microsoft Learn, Azure Architecture Center," Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/overview. [Data uzyskania dostępu: 24 kwietnia 2026].

[107] Microsoft, „Domain events: Design and implementation, Microsoft Learn," Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation. [Data uzyskania dostępu: 25 kwietnia 2026].

[108] G. Young, „CQRS Documents," 05 2026. [Online]. Available: https://cqrs.wordpress.com/wp-content/uploads/2010/11/cqrs_documents.pdf. [Data uzyskania dostępu: 24 kwietnia 2026].

[109] M. Fowler, „martinfowler.com,” 05 2026. [Online]. Available: https://martinfowler.com/bliki/CQRS.html. [Data uzyskania dostępu: 14 czerwca 2026].

[110] Microsoft, „CQRS Pattern, Microsoft Learn, Azure Architecture Center,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs. [Data uzyskania dostępu: 10 czerwca 2026].

[111] M. Fowler, „martinfowler.com,” 05 2026. [Online]. Available: https://martinfowler.com/eaaCatalog/dataTransferObject.html. [Data uzyskania dostępu: 30 maja 2026].

[112] Microsoft, „Microsoft Learn - Implementing the infrastructure persistence layer with Entity Framework Core,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-implementation-entity-framework-core. [Data uzyskania dostępu: 9 czerwca 2026].

[113] M. Fowler, „martinfowler.com,” 05 2026. [Online]. Available: https://martinfowler.com/eaaCatalog/unitOfWork.html. [Data uzyskania dostępu: 22 kwietnia 2026].

[114] C. Richardson, „Pattern: Transactional outbox, Microservices.io,” 05 2026. [Online]. Available: https://microservices.io/patterns/data/transactional-outbox.html. [Data uzyskania dostępu: 31 maja 2026].

[115] Microsoft, „Overview of ASP.NET Core authentication, Microsoft Learn,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/?view=aspnetcore-10.0. [Data uzyskania dostępu: 10 maja 2026].

[116] Microsoft, „Overview of ASP.NET Core authorization, Microsoft Learn,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/authorization/introduction?view=aspnetcore-10.0. [Data uzyskania dostępu: 13 czerwca 2026].

[117] OpenTelemetry, „OpenTelemetry - Getting Started for .NET, OpenTelemetry Documentation,” Cloud Native Computing Foundation, 05 2026. [Online]. Available: https://opentelemetry.io/docs/languages/dotnet/getting-started/. [Data uzyskania dostępu: 7 maja 2026].

[118] Hangfire, „Performing recurrent tasks, Hangfire Documentation,” Hangfire OÜ, 05 2026. [Online]. Available: https://docs.hangfire.io/en/latest/background-methods/performing-recurrent-tasks.html. [Data uzyskania dostępu: 16 maja 2026].

[119] Microsoft, „Worker services in .NET, Microsoft Learn,” Microsoft, 05 2026. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/extensions/workers. [Data uzyskania dostępu: 19 maja 2026].

[120] [Online]. Available: https://developers.cloudflare.com/ssl/edge-certificates/changing-dcv-method/. [Data uzyskania dostępu: 22 kwietnia 2026].

[121] [Online]. Available: https://developers.cloudflare.com/cloudflare-for-platforms/cloudflare-for-saas/. [Data uzyskania dostępu: 20 maja 2026].

[122] [Online]. Available: https://developers.cloudflare.com/api/resources/zero_trust/subresources/tunnels/subresources/cloudflared/subresources/configurations. [Data uzyskania dostępu: 24 maja 2026].

[123] [Online]. Available: https://developers.cloudflare.com/api/resources/custom_hostnames/. [Data uzyskania dostępu: 14 czerwca 2026].

[124] [Online]. Available: https://vpic.nhtsa.dot.gov/api/. [Data uzyskania dostępu: 3 czerwca 2026].

[125] [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html. [Data uzyskania dostępu: 18 czerwca 2026].

[126] [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_elements. [Data uzyskania dostępu: 12 maja 2026].

[127] [Online]. Available: https://v20.material.angular.dev/guide/creating-a-custom-form-field-control. [Data uzyskania dostępu: 19 kwietnia 2026].

[128] [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date. [Data uzyskania dostępu: 29 maja 2026].

[129] [Online]. Available: https://angular.dev/reference/versions. [Data uzyskania dostępu: 14 maja 2026].

[130] [Online]. Available: https://v20.material.angular.dev/components/datepicker/overview#internationalization. [Data uzyskania dostępu: 27 kwietnia 2026].

[131] [Online]. Available: https://v20.material.angular.dev/components/timepicker/overview#integration-with-matdatepicker. [Data uzyskania dostępu: 29 kwietnia 2026].

[132] [Online]. Available: https://github.com/juanjotorres90/ngx-material-intl-tel-input#readme. [Data uzyskania dostępu: 3 maja 2026].

[133] [Online]. Available: https://angular.dev/guide/forms/signals/async-operations#when-to-use-async-validation. [Data uzyskania dostępu: 7 czerwca 2026].

[134] [Online]. Available: https://rxjs.dev/guide/observable. [Data uzyskania dostępu: 5 maja 2026].

[135] [Online]. Available: https://rxjs.dev/guide/subject. [Data uzyskania dostępu: 24 maja 2026].

[136] [Online]. Available: https://rxjs.dev/api/index/class/BehaviorSubject. [Data uzyskania dostępu: 1 czerwca 2026].

[137] [Online]. Available: https://v20.angular.dev/api/core/AfterViewInit. [Data uzyskania dostępu: 11 czerwca 2026].

[138] [Online]. Available: https://rxjs.dev/api/index/function/combineLatest. [Data uzyskania dostępu: 4 maja 2026].

[139] [Online]. Available: https://rxjs.dev/api/operators/switchMap. [Data uzyskania dostępu: 11 maja 2026].

[140] Cure53, „dompurify,” 3 Czerwiec 2026. [Online]. Available: https://www.npmjs.com/package/dompurify. [Data uzyskania dostępu: 25 kwietnia 2026].

[141] [Online]. Available: https://enova.pl/moduly-systemu. [Data uzyskania dostępu: 18 maja 2026].

[142] [Online]. Available: https://dok.enova.pl/programowanie/architektura,4362. [Data uzyskania dostępu: 18 maja 2026].

[143] [Online]. Available: https://dok.enova.pl/programowanie/sesja-jako-kontener-danych,4370. [Data uzyskania dostępu: 16 czerwca 2026].

[144] [Online]. Available: https://dok.enova.pl/programowanie/business-xml,4368. [Data uzyskania dostępu: 24 czerwca 2026].

[145] [Online]. Available: https://dok.enova.pl/programowanie/worker-od-podstaw,4216. [Data uzyskania dostępu: 31 maja 2026].

[146] [Online]. Available: https://dok.enova.pl/programowanie/rozszerzanie-tabeli,4307. [Data uzyskania dostępu: 21 kwietnia 2026].

[147] M. Trzaska, Modelowanie i implementacja systemów informatycznych 2.0, ISBN 978-83-976442-0-5, 2025, p. 487.

[148] jaktestowac.pl, „jaktestowac.pl,” [Online]. Available: https://jaktestowac.pl/lesson/pw1s03l02/. [Data uzyskania dostępu: 18 czerwca 2026].

[149] Microsoft, „GitHub Copilot,” GitHub, 06 2026. [Online]. Available: https://github.com/features/copilot. [Data uzyskania dostępu: 18 maja 2026].

[150] Anysphere, „Cursor,” 06 2026. [Online]. Available: https://cursor.com. [Data uzyskania dostępu: 17 kwietnia 2026].

[151] Anthropic, „Claude Code,” 06 2026. [Online]. Available: https://www.anthropic.com/product/claude-code. [Data uzyskania dostępu: 11 czerwca 2026].

[152] Anthropic, „Claude Cowork,” 06 2026. [Online]. Available: https://www.anthropic.com/product/claude-cowork. [Data uzyskania dostępu: 28 maja 2026].

[153] Anthropic, „Claude,” 06 2026. [Online]. Available: https://claude.ai. [Data uzyskania dostępu: 25 kwietnia 2026].

[154] Google LLC, „Google Gemini,” 06 2026. [Online]. Available: https://gemini.google.com. [Data uzyskania dostępu: 11 czerwca 2026].

[155] OpenAI, „ChatGPT,” 06 2026. [Online]. Available: https://openai.com/chatgpt. [Data uzyskania dostępu: 9 maja 2026].

[156] OpenAI, „Codex,” 06 2026. [Online]. Available: https://openai.com/codex/. [Data uzyskania dostępu: 17 maja 2026].

[157] SpeakLeash & ACK Cyfronet AGH, „Bielik-11B-v.3.0-Instruct,” 06 2026. [Online]. Available: https://huggingface.co/speakleash/Bielik-11B-v3.0-Instruct. [Data uzyskania dostępu: 10 maja 2026].

[158] Element Labs, „LM Studio,” 06 2026. [Online]. Available: https://lmstudio.ai. [Data uzyskania dostępu: 9 maja 2026].

[159] StackOverflow, „AI | 2025 Developer Survey,” 2025. [Online]. Available: https://survey.stackoverflow.co/2025/ai. [Data uzyskania dostępu: 21 czerwca 2026].

[160] stackOverflow, „AI | 2024 Developer Survey,” 2024. [Online]. Available: https://survey.stackoverflow.co/2024/ai. [Data uzyskania dostępu: 21 czerwca 2026].

[161] N. Sakimura, J. Bradley i M. B. Jones, „JSON Web Token (JWT), RFC 7519,” Internet Engineering Task Force, 05 2026. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7519. [Data uzyskania dostępu: 4 maja 2026].