



POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Piotr Natkaniec – s28145

Tomasz Gajewski – s27751

Kacper Barański – s28166

Elastyczny i skalowalny system administracyjno- inwentyzacyjny do zarządzania środkami w przedsiębiorstwach

Praca inżynierska napisana pod
kierunkiem:
dr. inż. Mariusza Trzaski

Warszawa, lipiec 2026

.....
(data, miejscowość)

OŚWIADCZENIE STUDENTA DOTYCZĄCE WYKORZYSTANIA SZTUCZNEJ INTELIGENCJI (AI)

Ja, niżej podpisany/a:
(imię i nazwisko)

student/ka kierunku:
(nazwa kierunku)

numer albumu studenta:

Oświadczam, że zapoznałem/am 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.

Oświadczam, że ewentualne wykorzystanie narzędzi sztucznej inteligencji (AI) w niniejszej pracy miało charakter wyłącznie pomocniczy i nie stanowiło zastępstwa dla mojej samodzielnej pracy.

Oświadczam, że treść pracy stanowi moje własne, samodzielne opracowanie oraz ponoszę pełną odpowiedzialność za jej zawartość, niezależnie od wykorzystania narzędzi AI.

Oświadczam, że nie wykorzystywałem/am narzędzi AI do generowania treści stanowiących zasadniczą część pracy, w szczególności analiz, wniosków, interpretacji ani części podlegających ocenie jako przejaw twórczości własnej.

W przypadku korzystania z narzędzi AI sporządziłem/am Raport użycia AI zgodnie z obowiązującymi Wytycznymi i dołączyłem/am go do pracy.

.....

Streszczenie

Celem niniejszej pracy było stworzenie elastycznego i skalowalnego systemu administracyjno-inwentaryzacyjnego, przeznaczonego do zarządzania środkami w przedsiębiorstwach. System umożliwia przechowywanie środków, faktur, lokalizacji, kampanii inwentaryzacyjnych oraz obsługę incydentów i niestandardowych pól dla środków. Aplikacja została zaprojektowana w architekturze mikroservisowej, w której każda z aplikacji ma swoje ściśle określone zadania. Warstwa *backendowa* została zbudowana w oparciu o język Java w wersji 25 z wykorzystaniem *frameworka* Spring Boot, natomiast *frontend* zrealizowano jako aplikację SPA przy użyciu *frameworka* Next.js z językiem TypeScript.

Słowa kluczowe:

Java 25, Spring Boot, Next.js, Keycloak, RabbitMQ, AMS, ERP, PostgreSQL, Docker, konteneryzacja, aplikacja webowa.

Spis treści

1.	WSTĘP	7
1.1.	Cel i zakres pracy	7
1.2.	Organizacja pracy przy prototypie	8
1.3.	Struktura organizacyjna zespołu	11
2.	PRZEGLĄD I OCENA ISTNIEJĄCYCH ROZWIĄZAŃ	14
2.1.	Snipe IT	14
2.2.	GLPI	17
2.3.	EZO Inventory	18
2.4.	Asset Panda	20
3.	PROPOZYCJA SYSTEMU GLOWING WAFFLE ASSET MANAGEMENT SYSTEM	23
3.1.	Wizja systemu	23
3.2.	Założenia	23
3.3.	Słownik pojęć	24
3.4.	Role użytkowników i model uprawnień	25
3.5.	Wymagania funkcjonalne	26
3.6.	Aktorzy systemu	34
3.7.	Przypadki użycia	35
3.8.	Wymagania нефункционалне	41
3.9.	Projektowy diagram klas	41
3.9.1	Asset Service	41
3.9.2	Inventory Service	42
3.9.3	Loan Service	43
3.9.4	Incident Service	44
3.9.5	Reporting Service	45
3.10.	Diagramy stanów	47
3.11.	Diagramy aktywności	49
3.12.	Scenariusze	51
4.	TECHNOLOGIE ORAZ NARZĘDZIA	53
4.1.	Java 25	53
4.2.	Spring	54
4.3.	Hibernate	55
4.4.	Maven	55
4.5.	Baza danych PostgreSQL	56
4.6.	TypeScript	56
4.7.	Next.js	57
4.8.	RabbitMQ	57
4.9.	Docker	58
4.10.	Traefik	59
4.11.	Keycloak	60
4.12.	Git	61
4.13.	Github Actions	61
4.14.	Postman / Insomnia	62
4.15.	Discord	63
4.16.	Swagger	63
4.17.	OpenTelemetry	64
4.18.	SeaweedFS	64
4.19.	Prometheus	65
4.20.	Grafana	65
4.21.	Jaeger	66
4.22.	Grafana Loki	66
5.	DECYZJE ARCHITEKTONICZNE	67
5.1.	Uzasadnienie wyboru architektury mikroserwisowej	69

5.2.	Komunikacja asynchroniczna przy użyciu brokera wiadomości	72
5.3.	Komunikacja synchroniczna REST	75
5.4.	Serwer autoryzacji i autentykacji	76
5.5.	Warstwa bazodanowa	80
5.6.	Składowanie plików	82
5.7.	Projekt monitoringu i obserwowalności	85
6.	IMPLEMENTACJA SYSTEMU	91
6.1.	Wzorce projektowe użyte w systemie	91
6.1.1	Wzorzec specyfikacji	91
6.1.2	Wzorzec adaptera	93
6.1.3	Wzorzec fasady	94
6.1.4	Wzorzec Obserwator	95
6.2.	Implementacja infrastruktury serwerowej – backend	95
6.2.1	Struktura projektu	96
6.2.2	Architektura MVC	97
6.2.3	Encje i trwałość	104
6.2.4	Weryfikacja danych przychodzących oraz obsługa błędów	107
6.2.5	Konfiguracja brokera RabbitMQ i komunikacja oparta na zdarzeniach	110
6.3.	Implementacja klienckiej aplikacji webowej – frontend	116
6.3.1	Stylizacja i system komponentów UI	116
6.3.2	Struktura projektu	117
6.3.3	Definicje typów TypeScript	119
6.3.4	Komunikacja z mikroservisami backendowymi	120
6.3.5	Zarządzaniem stanem aplikacji	122
6.3.6	Tabele danych z TanStack Table	123
6.3.7	Uwierzytelnienie i zarządzanie sesją	125
6.4.	Mechanizmy zapewniające bezpieczeństwo	126
6.4.1	Katalog ról i moduł współdzielony	126
6.4.2	Konfiguracja Spring Security i walidacja tokenów JWT	127
6.4.3	Konwerter ról Keycloak i autoryzacja na poziomie metod	127
6.4.4	Konfiguracja CORS	129
6.4.5	Ochrona tras i warunkowe renderowanie po stronie aplikacji klienckiej	130
6.5.	Konteneryzacja i orkiestracja środowiska – DevOps	132
6.5.1	Wieloetapowe budowanie obrazów kontenerów	132
6.5.2	Orkiestracja środowiska za pomocą mechanizmu Docker Compose	134
6.5.3	Segmentacja wirtualnej sieci kontenerowej	135
6.5.4	Potok CI/CD	136
7.	INTERFEJS UŻYTKOWNIKA	139
7.1.	Logowanie i strona główna	139
7.2.	Zarządzanie środkami trwałymi	141
7.2.1	Widoki konfiguracyjne systemu ewidencji	142
7.2.2	Lista środków trwałych	146
7.2.3	Dodawanie i edycja środków trwałych	147
7.2.4	Widok szczegółów środków trwałych	148
7.3.	Kampanie inwentaryzacyjne	151
7.3.1	Lista kampanii	151
7.3.2	Cykl życia kampanii i widok szczegółów	152
7.3.3	Skanowanie środka trwałego w trakcie kampanii	153
7.3.4	Widok szczegółów rozbieżności	154
7.4.	Wypożyczenia	155
7.4.1	Wnioski o wypożyczenie	156
7.4.2	Decyzje	157
7.4.3	Transfery	157
7.4.4	Finalizowane wnioski	157
7.5.	Incydenty	158
7.5.1	Lista incydentów	158

7.5.2	<i>Widok szczegółów incydentu</i>	159
7.6.	Personel.....	160
7.7.	Widoki osobiste	162
7.7.1	<i>Moje środki trwałe</i>	163
7.7.2	<i>Moje incydenty</i>	163
7.7.3	<i>Moje wnioski o wypożyczenie</i>	164
7.7.4	<i>Moje kampanie inwentaryzacyjne</i>	165
8.	TESTOWANIE APLIKACJI	167
8.1.	Testy manualne	167
8.2.	Testy zautomatyzowane.....	170
9.	PODSUMOWANIE I WNIOSKI	171
9.1.	Ocena spełnienia wymagań funkcjonalnych i niefunkcjonalnych	171
9.1.1	<i>Spełnialność wymagań funkcjonalnych</i>	171
9.1.2	<i>Spełnialność wymagań niefunkcjonalnych</i>	172
9.2.	Napotkane problemy	174
9.3.	Dalszy rozwój	176
9.4.	Wnioski końcowe.....	177
10.	BIBLIOGRAFIA	178
11.	SPIS ILUSTRACJI	181
12.	SPIS TABEL	185
13.	SPIS LISTINGÓW	186

1. Wstęp

Rosnąca liczba środków trwałych w organizacjach sprawia, że ręczne prowadzenie ewidencji staje się nieefektywne i jest podatne na błędy. Niniejsza praca przedstawia propozycję systemu informatycznego, który adresuje ten problem i stara się wypełnić lukę obecną na rynku. W tym rozdziale omówiono kolejno cel i zakres pracy, organizację prac nad prototypem oraz strukturę organizacyjną zespołu odpowiedzialnego za pracę oraz prototyp.

1.1. Cel i zakres pracy

Celem niniejszej pracy było zaprojektowanie i zaimplementowanie elastycznego i skalowalnego systemu administracyjno-inwentaryzacyjnego, przeznaczonego do zarządzania środkami trwałymi w przedsiębiorstwach i instytucjach o średniej strukturze organizacyjnej. System ma umożliwiać prowadzenie ewidencji zasobów oraz powiązanych z nimi danych – faktur, lokalizacji, kampanii inwentaryzacyjnych, incydentów oraz wypożyczeń – przy zachowaniu możliwości elastycznego modelowania dziedziny za pomocą definiowanych przez użytkownika kategorii, typów oraz pól niestandardowych.

Motyacją do podjęcia tematu jest zaobserwowana luka rynkowa pomiędzy prostymi narzędziami inwentaryzacyjnymi, cechującymi się ograniczonymi możliwościami adaptacji przechowywanych danych, a rozbudowanymi systemami klasy ERP, których wdrożenie i utrzymanie wiąże się z wysokim kosztem oraz znaczną złożonością. Projektowany system ma zagospodarować tę przestrzeń, oferując elastyczny model danych oraz szeroki zakres obsługiwanych procesów przy jednoczesnym zachowaniu niskiego progu wdrożeniowego, łatwości utrzymania i niskich kosztów. Szersze omówienie adresowanych luk omówiono w porównaniu do rozwiązań konkurencyjnych w rozdziale 2, a wizję systemu i jego propozycję przedstawiono w rozdziale 3.

Tak sformułowany cel główny przekłada się na następujące cele szczegółowe:

- przegląd i krytyczna ocena istniejących rozwiązań rynkowych klasy AMS oraz ITAM, w celu osadzenia projektowanego systemu w kontekście rynkowym i zidentyfikowania jego cech wyróżniających;
- zdefiniowanie wymagań funkcjonalnych i нефункциональных systemu wraz z modelem ról użytkowników i uprawnień;
- zaprojektowanie architektury systemu w oparciu o wzorzec mikroservisowy wraz z uzasadnieniem przyjętych decyzji architektonicznych;
- implementacja warstwy backendowej w języku Java z wykorzystaniem frameworka Spring Boot, w postaci zbioru niezależnych mikroservisów komunikujących się asynchronicznie oraz synchronicznie;
- implementacja klienckiej aplikacji webowej w formie aplikacji SPA we frameworku Next.js z językiem TypeScript;
- zaprojektowanie elastycznego modelu danych umożliwiającego rozszerzanie ewidencji o atrybuty specyficzne dla danej organizacji;
- integracja systemu z zewnętrznym dostawcą tożsamości oraz katalogiem LDAP w oparciu o standardy OAuth 2.0 oraz OpenID Connect;
- przygotowanie środowisk uruchomieniowych oraz warstwy obserwowalności systemu z wykorzystaniem konteneryzacji i otwartych standardów danych telemetrycznych.

Realizacja powyższych celów prowadzi do rezultatu, którym jest w pełni funkcjonalny prototyp systemu Glowing Waffle Asset Management System, stanowiący praktyczną weryfikację przyjętych założeń projektowych oraz architektonicznych. Niniejsza praca nie ma na celu dostarczenia gotowego produktu komercyjnego, lecz wykazanie zasadności i wykonalności przyjętej koncepcji; ograniczenia prototypu oraz kierunki dalszego rozwoju omówiono w rozdziale 9.

1.2. Organizacja pracy przy prototypie

Wspólnie ustalono, że na potrzeby prototypu praca będzie prowadzona w trybie przyrostowym i iteracyjnym. Zamiast jednorazowej realizacji całości systemu, poszczególne jego moduły były realizowane samodzielnie i stopniowo, możliwie małymi, wydzielonymi częściami funkcjonalności. Taki tryb pracy najlepiej wpisywał się w doświadczenie zespołu i potrzebę „zgrania” ze sobą wytwarzania backendu i frontendu równolegle.

Kod źródłowy całego systemu przechowywano we wspólnym repozytorium na platformie GitHub, obejmującym sześć mikroservisów backendu, bibliotekę współdzieloną, aplikację frontendową oraz konfigurację infrastruktury i środowisk uruchomieniowych. Główny widok repozytorium na platformie GitHub przedstawiono na Rysunek 1.

The screenshot displays the GitHub interface for the repository 'glowing-waffle' (Private). The repository has 29 branches, 0 tags, 1 watch, 0 forks, and 1 star. The main content area shows a list of files and folders with their commit history:

File/Folder	Description	Commit Time
.github/workflows	Introduce reporting service on PROD	2 months ago
.mvn/wrapper	Repository created, spring boot added	8 months ago
ad	Prod deploy pipeline job, fixes in fronten...	6 months ago
ams-shared-api	synchronize responsibleUnitId with loan ...	2 months ago
asset-service	stop sending empty update events to re...	2 months ago
frontend-service	FE bump dependencies, change dockerfil...	16 hours ago
incident-service	bump otel library to resolve CVE-2026-3...	2 months ago
inventory-service	bump otel library to resolve CVE-2026-3...	2 months ago
kc-proxy	bump otel library to resolve CVE-2026-3...	2 months ago
keycloak	Implement loan management service wit...	3 months ago
loan-service	fix for handling situations where loan is ...	2 months ago
monitoring	Open Telemetry amendments	6 months ago
notification-service	bump otel library to resolve CVE-2026-3...	2 months ago
rabbitmq	Add user existence verification during up...	3 months ago
reporting-service	bump otel library to resolve CVE-2026-3...	2 months ago
seaweedfs	lfeat: Core incident service logic. Fixed tr...	5 months ago
src	Repository created, spring boot added	8 months ago
.dockerignore	Add traceparent header propagation for ...	3 months ago
.gitattributes	Repository created, spring boot added	8 months ago
.gitignore	add initial database schema and entities	6 months ago
Dockerfile	Modules	8 months ago
HELP.md	Repository created, spring boot added	8 months ago
README.md	Switched entity identifiers from Integer t...	7 months ago

The sidebar on the right contains the following sections:

- About:** Engineering Thesis, Readme, Activity, 1 star, 1 watching, 0 forks.
- Releases:** No releases published, [Create a new release](#).
- Packages:** 8 packages listed: glowing-waffle/notification-service, glowing-waffle/inventory-service, glowing-waffle/asset-service, and + 5 packages.
- Contributors:** 3 contributors: Patkaniec22 (Piotr Natkaniec), s28166 (Kacper Barański), and TGajewski09 (s27751).
- Languages:** TypeScript 63.0%, Java 35.8%, Other 1.2%.

Rysunek 1 Główny widok repozytorium prototypu Glowing Waffle AMS na platformie GitHub. Źródło: Opracowanie własne.

Przyjęto model pracy oparty na tzw. gałęziach funkcjonalnych (ang. *feature branch workflow*), w którym każda nowa funkcjonalność lub poprawka rozwijana była na odrębnej gałęzi, odgałęzionej od gałęzi głównej main. Warto wspomnieć, że według [1] taki model pracy w Git jest jednym z możliwych i najczęściej wybieranych modeli, ale nie jest jedynym dostępnym trybem. Zastosowano czytelną konwencję nazewnictwa gałęzi, odzwierciedlającą charakter prac, m.in. feature/ dla nowych funkcjonalności, fix/ dla poprawek błędów oraz refactor/ dla refaktoryzacji kodu. Listę niektórych gałęzi przedstawiono na Rysunek 2.

Branches

New branch

Overview

Yours

Active

Stale

All

Q Search branches...

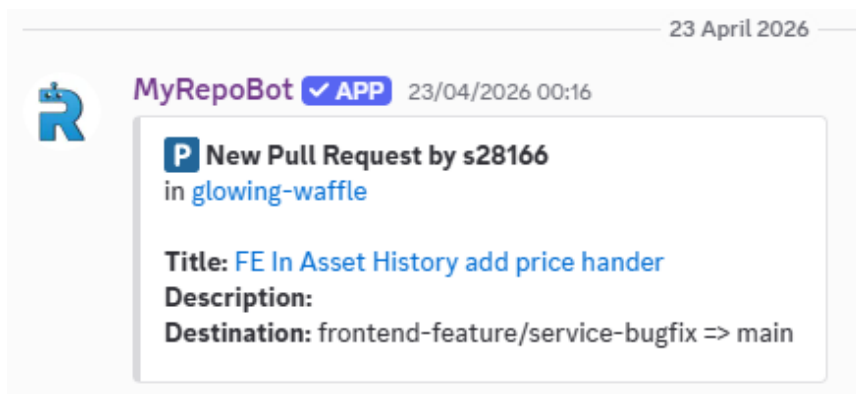
Branch	Updated	Check status	Behind	Ahead	Pull request	
main	16 hours ago	8 / 8		Default		
frontend-feature/post-implementation...	16 hours ago	1 / 1	2	0	#87	
refactor/authorization-role	17 hours ago		9	12		
frontend-feature/post-acceptance-cha...	2 months ago	8 / 8	9	0		
feature/initializers	2 months ago	1 / 1	10	0	#83	
frontend-feature/service-bugfix	2 months ago	1 / 1	16	0	#82	
feature/enhanced-loan	2 months ago		46	0		
fix/fast-fixes	2 months ago	1 / 1	97	0	#77	
feature/asset-history	2 months ago		63	0		
feature/global-settings-framework	2 months ago		88	0	#67	
refactors/n1	2 months ago	1 / 1	101	0	#70	
feature/loan-scheduler	2 months ago	1 / 1	107	0	#68	
frontend-feature/service-improvements	2 months ago		111	0	#66	

Rysunek 2 Lista niektórych gałęzi rozwijania kodu, które powstały podczas tworzenia prototypu. Źródło: Opracowanie własne.

Łączenie zmian z gałęzią główną odbywało się wyłącznie poprzez mechanizm PR, które polega na scalaniu zmian, i wymagało akceptacji dwóch członków zespołu oraz przeprowadzenia procesu sprawdzenia kodu. Ustanowiło to naturalne miejsce kontroli jakości oraz zapewniało wymianę wiedzy o poszczególnych częściach systemu. Dopiero zaakceptowane żądanie scalano z gałęzią main, traktowaną jako gałąź stabilną, będąca docelową gałęzią dla procesu wdrożeniowego. W trakcie realizacji projektu w ten sposób obsłużono blisko dziewięćdziesiąt żądań PR.

Procesy budowania i wdrażania zautomatyzowano przy pomocy mechanizmu GitHub Actions (podrozdział 4.13), realizującego koncepcję ciągłej integracji i ciągłego dostarczania (ang. *Continuous Integration / Continuous Delivery*, CI/CD). Po scaleniu zmian do gałęzi main uruchamiany jest potok, który równolegle buduje obrazy kontenerowe (podrozdział 4.9) wszystkich serwisów systemu i publikuje je do rejestru obrazów GitHub Container Registry, skąd są one następnie wykorzystywane w procesie wdrożenia na środowisko produkcyjne. Dzięki temu droga od zaakceptowanej zmiany w kodzie do działającej wersji systemu pozostaje powtarzalna i w znacznym stopniu zautomatyzowana.

Bieżącą komunikację w zespole prowadzono z wykorzystaniem komunikatora Discord (podrozdział 4.15), który pełnił rolę głównego kanału koordynacji prac oraz omawiania decyzji projektowych. Platformę tę zintegrowano z platformą Github w taki sposób, że każdy nowy otwarty PR skutkował powiadomieniem na dedykowanym kanale platformy Discord. Usprawniło to komunikację w zespole i sposób informowania członków w zespole o potrzebie przejrzania zmian przed scaleniem do gałęzi main. Jedną z takich notyfikacji zaprezentowano na Rysunek 3.



Rysunek 3 Powiadomienie informujące o nowym, otwartym PR, wysłanym przez mechanizm integracyjny GitHub Discord MyRepoBot. Źródło: Opracowanie własne.

1.3. Struktura organizacyjna zespołu

Praca nad systemem Glowing Waffle AMS była realizowana zespołowo. Grupa składała się z trzech osób: Piotra Natkańca, Tomasza Gajewskiego oraz Kacpra Barańskiego. Z uwagi na mikroservisowy charakter systemu i wyraźny podział na warstwy infrastrukturalną, backendową oraz frontendową, przyjęto model pracy oparty na podziale odpowiedzialności według zadeklarowanych kompetencji, przy jednoczesnym zachowaniu wspólnego nadzoru nad spójnością i kierunkiem całego rozwiązania. Podział ten nie miał charakteru ścisłego rozgraniczenia, lecz wyznaczał funkcje obszary dla poszczególnych członków zespołu. Wszystkie kluczowe decyzje projektowe oraz architektoniczne były konsultowane wspólnie celem osiągnięcia jednomyślności w zakresie dalszych kroków.

Szczegółowy podział obowiązków w prototypie i autorstwo poszczególnych rozdziałów pracy inżynierskiej przedstawiono w Tabeli 1 i Tabeli 2.

Tabela 1 Rozdzielenie zakresu odpowiedzialności w pracy przy implementacji prototypu. Źródło: Opracowanie własne.

Imię i nazwisko	Zakres odpowiedzialności
Piotr Natkaniec	Koncepcja architektoniczna systemu i dobór stosu technologicznego (rozdz. 4-5); architektura mikroserwisowa z komunikacją synchroniczną i asynchroniczną (5.1-5.3); autoryzacja i uwierzytelnianie (5.4); warstwa bazodanowa (5.5); składowanie plików (5.6); obserwowalność oparta o OpenTelemetry (5.7); część infrastruktury serwerowa backendowej (6.2), konteneryzacja DevOps (6.5)
Kacper Barański	Całość warstwy frontendowej; aplikacja SPA w Next.js (TypeScript): kliencka aplikacja webowa z systemem komponentów UI, integracją z serwerem autoryzacji i komunikacją z backendem (6.3); projekt interfejsu użytkownika dla poszczególnych ról systemowych (rozdz. 7)
Tomasz Gajewski	Koncepcja biznesowa systemu, zaplanowanie rdzeniowych funkcjonalności systemu; logika dziedziczna mikroserwisów backendu (6.2) i zastosowane wzorce projektowe (6.1); modelowanie UML - przypadki użycia i aktorzy (3.6-3.7), wymagania funkcjonalne i niefunkcjonalne (3.5, 3.7), analityczny diagram klas (3.9), diagramy stanów (3.10) i diagramy aktywności (3.11)

Tabela 2 Szczegółowy rozkład autorstwa rozdziałów tekstu pracy inżynierskiej. Źródło: Opracowanie własne.

Imię i nazwisko	Napisane rozdziały
Piotr Natkaniec	Rozdział 1 (1, 1.1-1.3) Rozdział 2 (2, 2.1-2.4) Rozdział 3 (3.8, 3.9) Rozdział (5, 5.1-5.7) Rozdział 6 (6.2, 6.2.1-6.2.3, 6.2.5, 6.5) Rozdział 8 (8, 8.1-8.2) Rozdział 9 (9, 9.1-9.4)
Kacper Barański	Rozdział 3 (3.11) Rozdział 4 (4, 4.1-4.7, 4.9, 4.12, 4.13, 4.16-4.22) Rozdział 6 (6, 6.3, 6.4) Rozdział 7 (7, 7.1-7.7)
Tomasz Gajewski	Rozdział 3 (3, 3.1-3.7, 3.10, 3.12) Rozdział 4 (4.8, 4.10, 4.11, 4.14, 4.15) Rozdział 6 (6.1, 6.2.2, 6.2.4)

2. Przegląd i ocena istniejących rozwiązań

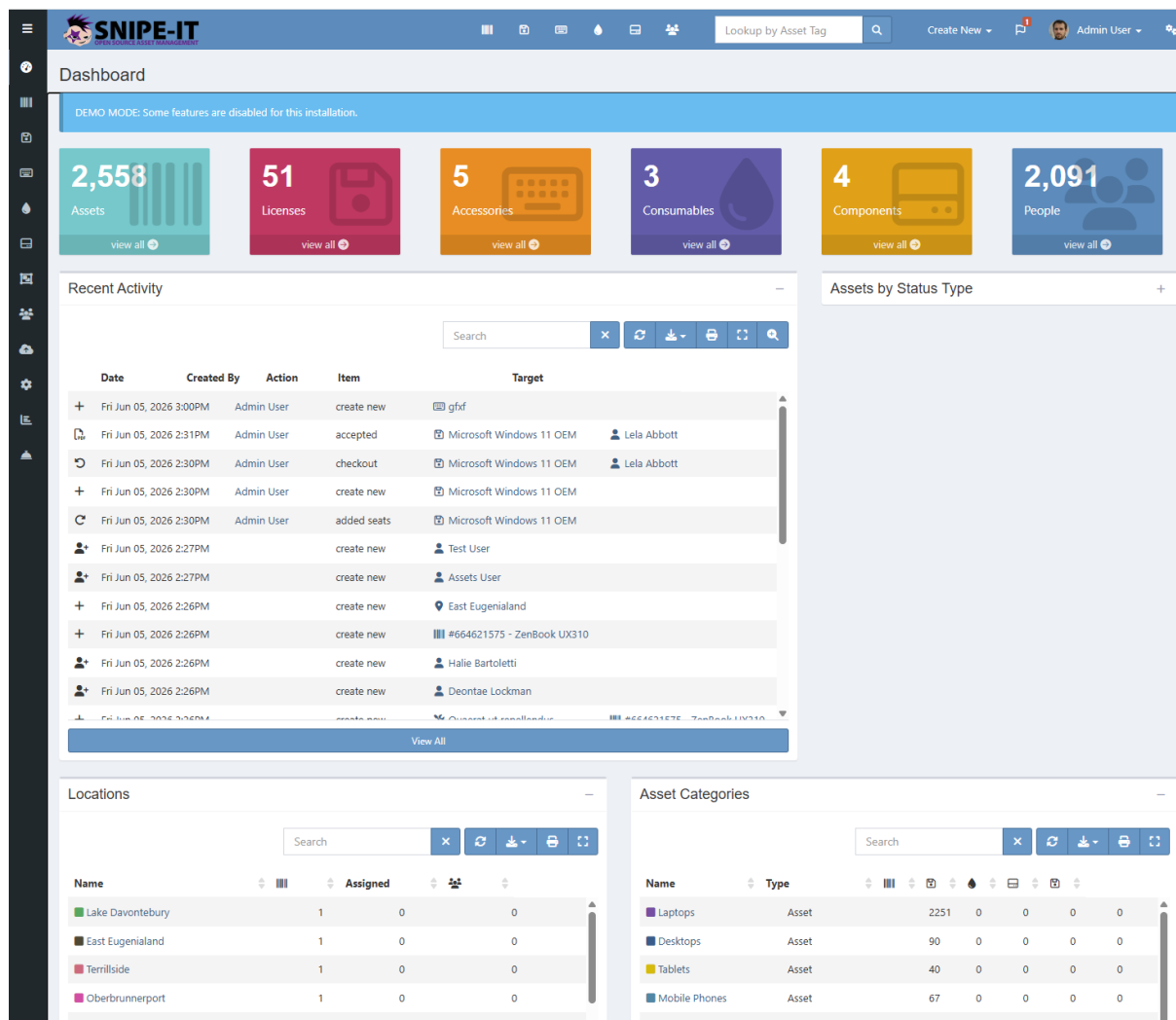
Niniejszy rozdział stanowi przegląd rozwiązań rynkowych klasy Asset Management System oraz ich krytyczną ocenę w odniesieniu do założeń przyjętych dla systemu Glowing Waffle AMS, które szerzej opisano w podrozdziale 3.1. Celem analizy jest osadzenie proponowanego systemu w kontekście rynkowym, zidentyfikowanie luki, którą ma on wypełniać, oraz wskazanie cech wyróżniających go na tle konkurencji. Celowo pominięto szczegółową analizę rozbudowanych platform korporacyjnych, takich jak ServiceNow, gdyż rząd wielkości takich narzędzi jest nieporównywalny do skali prototypu i jego umiejscowienia. Niemniej, przed przejściem do oceny konkretnych produktów należy uporządkować nazewnictwo, ponieważ rynek oprogramowania do zarządzania zasobami obejmuje kilka pokrewnych, lecz odmiennych klas systemów.

Najszerszym z omawianych pojęć jest system zarządzania zasobami typu AMS, rozumiany jako oprogramowanie wspierające ewidencję i nadzór nad cyklem życia zasobów organizacji. W jego obrębie wyróżnić można węższe specjalizacje. Systemy klasy ITAM (ang. *IT Asset Management*) ograniczają się zazwyczaj do zasobów informatycznych – sprzętu informatycznego, licencji oprogramowania oraz zasobów chmurowych – często łącząc się z obszarem zarządzania usługami IT (ang. *IT Service Management*, ITSM) oraz bazą konfiguracji (ang. *Configuration Management Database*, CMDB) [2]. Oprogramowanie do zarządzania środkami trwałymi (ang. *Fixed Asset Management Software*, FAMS) koncentruje się z kolei na księgowym ujęciu majątku przedsiębiorstwa, w tym na amortyzacji oraz ewidencji podatkowej. Systemy klasy EAM (ang. *Enterprise Asset Management*) oraz CMMS (ang. *Computerized Maintenance Management System*) kładą nacisk na utrzymanie ciągłości ruchu urządzeń i obsługę aktywów o wysokiej wartości, typowych dla środowisk przemysłowych. Na przeciwnym, najbardziej kompleksowym krańcu znajdują się systemy ERP (ang. *Enterprise Resource Planning*), w których zarządzania majątkiem jest jednym z wielu dostępnych modułów.

Na tle powyższej klasyfikacji system Glowing Waffle AMS plasuje się jako ogólny system klasy AMS, nieograniczony – w przeciwieństwie do rozwiązań ITAM – wyłącznie do zasobów informatycznych, a jednocześnie lżejszy i tańszy we wdrożeniu niż wielkoskalowe systemy ERP. W odróżnieniu od oprogramowania klasy FAMS nacisk położono na operacyjną ewidencję i inwentaryzację, nie zaś na rozliczenia księgowe, natomiast w przeciwieństwie do systemów EAM i CMMS utrzymanie ciągłości ruchu urządzeń nie stanowi funkcji centralnej, niemniej obsługa incydentów wykazuje pewne podobieństwo do obszaru ITSM.

2.1. Snipe IT

Snipe IT jest otwartoźródłowym systemem do zarządzania zasobami typu ITAM, rozwijanym przez firmę Grokability, Inc. Aplikacja udostępniana jest na licencji AGPL-3.0, a jej kod źródłowy zbudowano w oparciu o język PHP i framework Laravel w wersji 12, z wykorzystaniem relacyjnej bazy danych MySQL/MariaDB [3]. Rozwiązanie ma postać aplikacji webowej i może być wdrażane zarówno we własnej infrastrukturze bez opłat licencyjnych, jak i w płatnym wariantcie hostowanym typu SaaS. Pod względem pozycjonowania Snipe IT koncentruje się na ewidencji sprzętu komputerowego, licencji oprogramowania, akcesoriów oraz materiałów eksploatacyjnych, stanowiąc jedno z najpopularniejszych otwartoźródłowych narzędzi tej klasy [4]. Główny widok aplikacji zilustrowano na Rysunek 4.



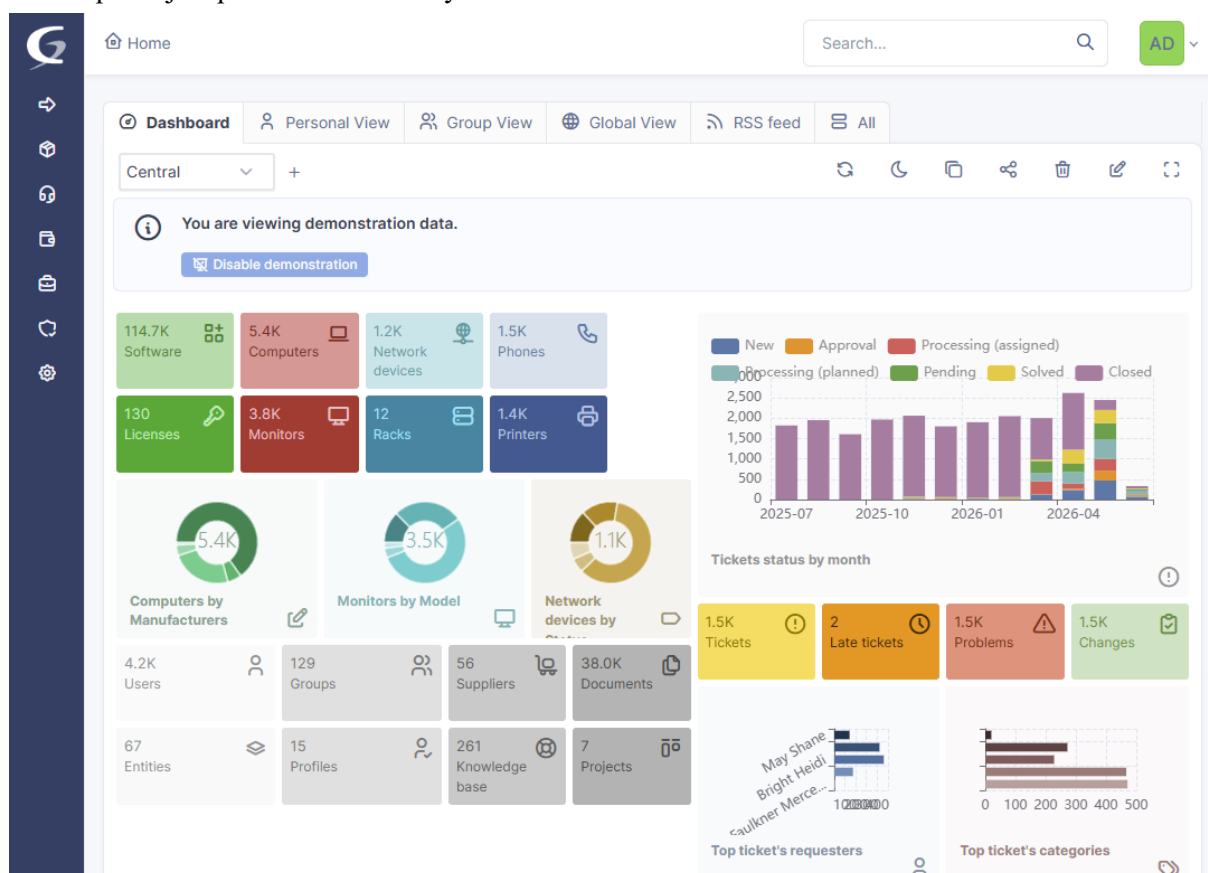
Rysunek 4 Główny widok użytkownika w wersji demonstracyjnej otwartoźródłowego oprogramowania Snipe-IT. Źródło: Opracowanie własne.

Model danych systemu opiera się na hierarchii obejmującej producentów, modele oraz kategorie, do których przypisywane są poszczególne egzemplarze zasobów; uzupełniają go lokalizacje, firmy oraz działy. Istotną z perspektywy niniejszej pracy cechą jest mechanizm pól niestandardowych, grupowanych w zestawy (ang. *fieldsets*), pozwalający na rozszerzenie modelu o atrybuty specyficzne dla danej dziedziny, takie jak numer seryjny czy parametry techniczne, co zaprezentowano na Rysunek 5.

wydawany użytkownikowi przez uprawnionego użytkownika systemu, brakuje natomiast samoobsługowego procesu wnioskowania i automatycznego wydawania środków, co realizowane jest przez system Glowing Waffle AMS. Po czwarte, Snipe IT nie posiada dedykowanego modułu zgłaszania i obsługi incydentów wraz z załącznikami oraz weryfikacją zgłoszeń, ani pełnego mechanizmu kampanii inwentaryzacyjnych z udziałem wielu pracowników i rejestrowaniem rozbieżności; udostępnia jedynie uproszczoną funkcję audytu zasobów oraz rejestrowania czynności serwisowych na zasadzie check-in/check-out. Wreszcie, z perspektywy architektonicznej Snipe IT stanowi aplikację monolityczną, co odróżnia go od mikroserwisowego podejścia przyjętego w niniejszej pracy. Reasumując, Snipe IT jest dojrzałym i godnym uwagi przedstawicielem otwartoźródłowych systemów klasy ITAM, jednak jego ukierunkowanie na zasoby IT oraz ograniczony zakres procesów dziedzicznych pozostawiają przestrzeń, którą system Glowing Waffle AMS stara się wypełnić.

2.2. GLPI

GLPI (fr. *Gestionnaire Libre de Parc Informatique*) jest otwartoźródłowym systemem łączącym zarządzanie ITAM z obszarem ITSM oraz bazą zarządzania konfiguracją CMDB. Projekt zapoczątkowano w 2003 roku, a jego rozwój prowadzony jest przez społeczność oraz firmę Teclib, pełniącą rolę wydawcy oprogramowania. Aplikacja udostępniana jest na licencji GPL 3.0, a jej kod źródłowy, podobnie jak w przypadku omówionego wcześniej Snipe IT, opiera się na języku PHP oraz relacyjnej bazie danych MySQL bądź MariaDB [5]. GLPI ma postać aplikacji webowej i może być wdrażany zarówno lokalnie, we własnej infrastrukturze organizacji, jak i w komercyjnym wariancie SaaS. W odróżnieniu od Snipe IT, ukierunkowanego przede wszystkim na samą ewidencję, GLPI jest rozwiązaniem znacznie szerszym, obejmującym pełny cykl obsługi zgłoszeń serwisowych. Główny widok aplikacji zaprezentowano na Rysunek 6.



Rysunek 6 Główny widok aplikacji GLPI, przedstawiający podsumowanie wszystkich encji w systemie.
Źródło: Opracowanie własne.

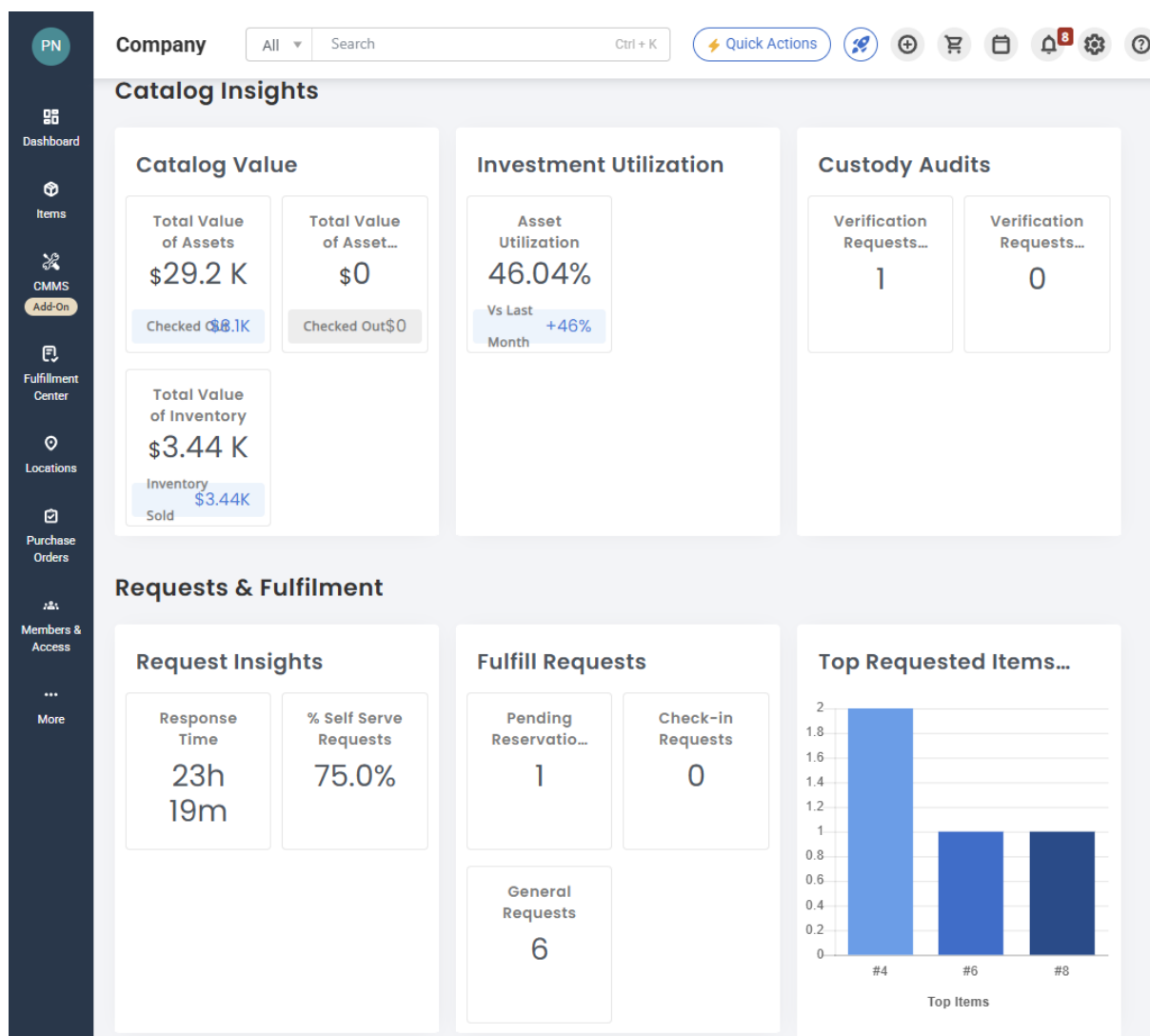
W warstwie ewidencyjnej system operuje na predefiniowanych typach obiektów, tj. komputerach, urządzeniach sieciowych, oprogramowaniu czy drukarkach. Encje te opisywane są atrybutami administracyjno-finansowymi, takimi jak dane zakupu, gwarancje oraz amortyzacja. Od wersji 10, zasoby te mogą być wykrywane automatycznie, za pośrednictwem dedykowanego agenta inwentaryzacyjnego skanującego sieć organizacji przy pomocy, między innymi, protokołu SNMP. Po stronie usługowej GLPI rozróżnia cztery kategorie zgłoszeń ITSM - incydenty, wnioski, problemy oraz zmiany, z definiowaniem warunków świadczenia usług informatycznych (ang. *Service Level Agreement*, SLA) i portalem samoobsługowym *self-service*, w którym użytkownik może zarezerwować sprzęt. Całość uzupełniają: wielopoziomowa struktura jednostek organizacyjnych, uprawnienia oparte na profilach, integracja z katalogiem LDAP oraz Active Directory wraz z logowaniem jednokrotnym, a także rozbudowany ekosystem wtyczek rozszerzających funkcjonalność narzędzia [5].

Na tle założeń systemu Glowing Waffle AMS, program GLPI wykazuje wyraźne podobieństwa. Jako rozwiązanie otwartoźródłowe, możliwe do wdrożenia lokalnie i zintegrowane z katalogiem LDAP, wpisuje się w przyjęty model wdrożenia oraz zarządzania tożsamością, a jego wbudowany moduł obsługi zdarzeń serwisowych stanowi odpowiednik projektowanego mechanizmu incydentów. Również funkcje finansowe, obejmujące zamówienia i gwarancje, są bliskie prowadzonej w proponowanym systemie ewidencji faktur. Cechy te, w połączeniu z ponad dwudziestoletnią dojrzałością projektu, czynią GLPI najpoważniejszym z analizowanych rozwiązań otwartoźródłowych.

Decydujące różnice uwidaczniają się jednak w przeznaczeniu narzędzia. GLPI pozostaje zorientowane na kontekst informatyczny, przez co ewidencja ogólnego majątku trwałego, np. mebli czy sprzętu AGD, wymaga obejść, a rozszerzanie sztywnego modelu danych o pola niestandardowe nie jest funkcją wbudowaną, lecz wymaga wtyczki. Odmienne jest też samo rozumienie inwentaryzacji, sprowadzonej w GLPI do automatycznego wykrywania urządzeń w sieci, nie zaś do planowej kampanii spisu z natury z rejestrowaniem rozbieżności. W zestawieniu z wysokim progiem wejścia, wynikającym z kompleksowości zgodnej z ITIL, lokuje to GLPI bliżej „cięższego i kompleksowego” krańca rynku i potwierdza zasadność budowy lżejszego, uogólnionego systemu, jakim jest Glowing Waffle AMS.

2.3. EZO Inventory

EZO Inventory jest komercyjnym, chmurowym systemem do zarządzania zasobami oraz inwentarzem, rozwijanym przez amerykańską firmę EZO, działającą wcześniej pod nazwą EZOfficeInventory. W odróżnieniu od omówionych wcześniej rozwiązań otwartoźródłowych, narzędzie udostępniane jest wyłącznie w modelu SaaS w ramach płatnej subskrypcji, której koszt uzależniony jest od wybranego planu oraz liczby ewidencjonowanych środków trwałych, a jego kod źródłowy pozostaje zamknięty [6]. Rozwiązanie ma postać aplikacji webowej uzupełnionej o dedykowane aplikacje mobilne na platformy iOS oraz Android, a pod względem pozycjonowania koncentruje się na śledzeniu sprzętu biurowego i informatycznego oraz na zarządzaniu materiałami eksploatacyjnymi. Główny panel aplikacji zilustrowano na Rysunek 7.



Rysunek 7 Główny kokpit administracyjny EZO Inventory. Źródło: Opracowanie własne.

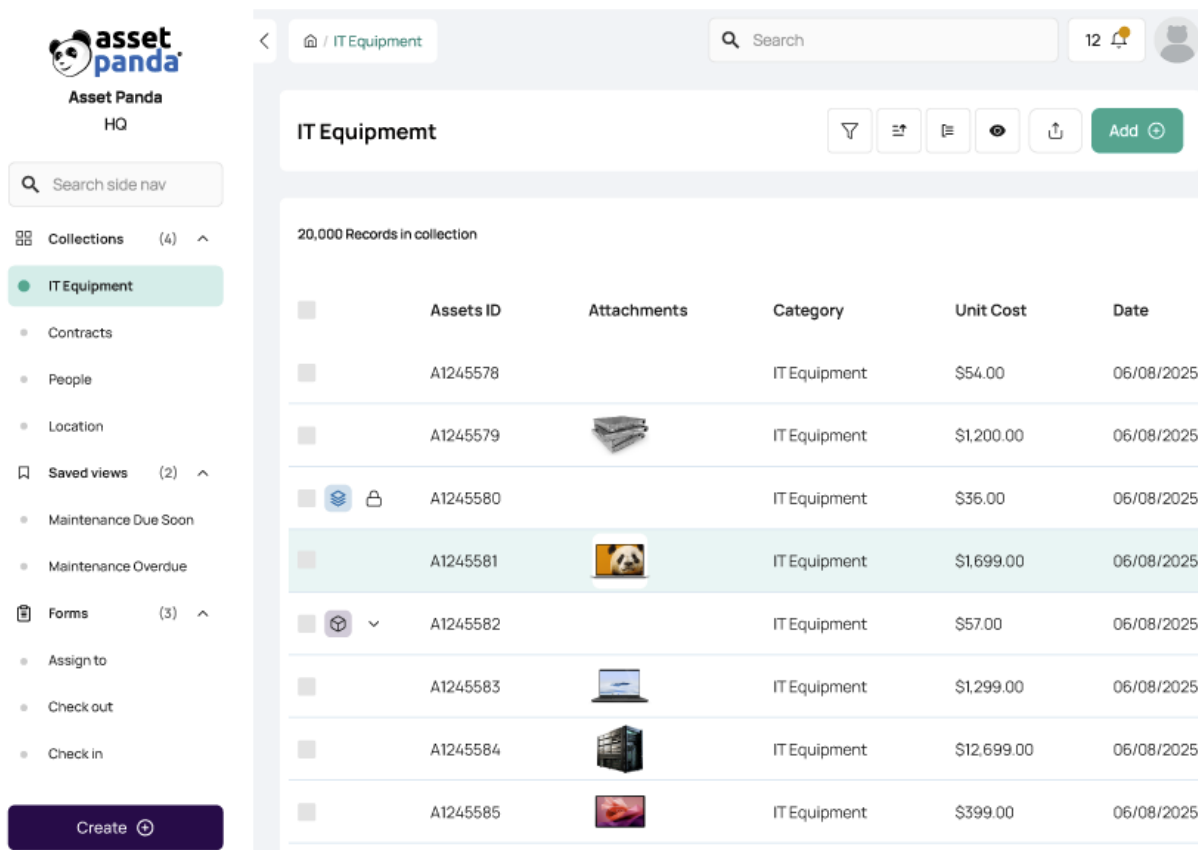
Model danych systemu opiera się na rekordach zasobów grupowanych w kategorie i opisywanych zestawem atrybutów, które można rozszerzać o pola niestandardowe. Narzędzie realizuje pełen cykl życia zasobu, w tym mechanizm wydania i zwrotu (ang. *check-in/check-out*), rezerwacje, ewidencję amortyzacji, obsługę przeglądów serwisowych oraz funkcję audytu stanu posiadania. Osobny moduł *inventory* pozwala na kontrolę zapasów materiałów zużywalnych (papier, toner do drukarki i tym podobne) wraz z powiadomieniami o niskim stanie magazynowym. Identyfikację środków wspiera generowanie etykiet z kodami kreskowymi, kodami QR oraz znacznikami RFID, których odczyt możliwy jest z poziomu aplikacji mobilnej. EZO Inventory umożliwia synchronizację z katalogiem LDAP oraz Active Directory, logowanie jednokrotne w standardzie SAML 2.0, a także współpracę z systemami zewnętrznymi za pośrednictwem interfejsu REST API [6].

Narzędzie EZO Inventory wykazuje kilka cech zbieżnych z przyjętą wizją. Mechanizm pól niestandardowych częściowo odpowiada wymogowi elastycznego modelowania dziedziny, integracja z katalogiem LDAP oraz logowaniem jednokrotnym pokrywa się z przyjętym modelem zarządzania tożsamością, a rozbudowane wsparcie dla kodów kreskowych i QR wraz z ich odczytem z urządzenia mobilnego jest bliskie zakładanej w prototypie obsłudze inwentaryzacji z poziomu telefonu. Dojrzałość komercyjna produktu, jego rozbudowane aplikacje mobilne oraz szeroki zakres funkcji operacyjnych czynią go reprezentatywnym przykładem chmurowych narzędzi tej klasy.

Zasadnicze różnice ujawniają się jednak w modelu wdrożenia i dostępności. EZO Inventory dostarczane jest wyłącznie w wariantcie chmurowym typu SaaS, co stoi w sprzeczności z przyjętym dla prototypu wymogiem wdrożenia lokalnego oraz zachowania pełnej kontroli organizacji nad własnymi danymi, natomiast rozliczany abonamentowo koszt utrzymania, rosnący wraz z liczbą ST, oddala je od zakładanego niskiego progu wdrożeniowego i kosztowego. Zamknięty charakter oprogramowania ogranicza możliwość jego modyfikacji oraz uzależnia organizację od zewnętrznego dostawcy. Niezależnie od ograniczeń modelu biznesowego, system nie udostępnia hierarchicznej struktury kategoria–typ z dziedziczeniem pól oraz przedrostków tagów, a obsługa wypożyczeń koncentruje się na administracyjnym wydaniu i zwrocie środków, nie zaś na pełnym, samoobsługowym procesie wnioskowania, zatwierdzania i automatycznego wydawania realizowanym w systemie Glowing Waffle AMS.

2.4. Asset Panda

Asset Panda jest komercyjną, chmurową platformą ITAM do zarządzania zasobami i śledzenia ich cyklu życia, rozwijaną przez amerykańską firmę o tej samej nazwie. Podobnie jak EZO Inventory, narzędzie udostępniane jest w modelu SaaS, a jego kod źródłowy pozostaje zamknięty; charakterystyczny dla tego rozwiązania jest model rozliczeń uzależniony od liczby ewidencjonowanych zasobów, przy nielimitowanej liczbie użytkowników [7]. Produkt pozycjonowany jest jako wysoce konfigurowalna platforma typu *no-code*, kładąca nacisk na możliwość dostosowania struktury danych oraz procesów do specyfiki organizacji bez konieczności programowania. Rozwiązanie ma postać aplikacji webowej uzupełnionej o aplikacje mobilne na platformy iOS oraz Android. Widok ekranu ze ST w aplikacji zaprezentowano na Rysunek 8; z powodu braku możliwości rejestracji konta w trybie demonstracyjnym, zrzut ekranu zaczerpnięto ze strony producenta.



Rysunek 8 Widok zarządzania środkami trwałymi w aplikacji Asset Panda. Źródło: [7]

Administrator może definiować własne pola, akcje oraz procesy pracy, dzięki czemu system da się dopasować do sposobu ewidencji majątku w danej firmie. Platforma wspiera obsługę cyklu życia zasobów, w tym wydawanie i zwroty, rezerwacje, zlecenia serwisowe, audyty oraz ewidencję amortyzacji liniowej. Do rekordów zasobów można dodawać załączniki, takie jak dokumenty i zdjęcia, a identyfikację środków trwałych ułatwia skanowanie kodów kreskowych oraz kodów QR w aplikacji mobilnej. Asset Panda oferuje także konfigurowalne raporty, role i uprawnienia użytkowników oraz integracje z innymi systemami, między innymi przez logowanie SAML, synchronizację z Active Directory oraz API [7].

Z punktu widzenia założeń systemu Glowing Waffle AMS, Asset Panda można uznać za jedno z rozwiązań komercyjnych najbardziej zbliżonych do idei elastycznego modelowania dziedziny. Platforma pozwala dostosowywać strukturę danych przez konfigurację pól oraz definiowanie akcji, co odpowiada potrzebie dopasowania ewidencji zasobów do procesów konkretnej organizacji. Możliwość dodawania załączników, takich jak zdjęcia i dokumenty, jest zbliżona do zakładanej w prototypie obsługi dokumentacji związanej z zasobami i incydentami. System oferuje również role oraz konfigurowalne uprawnienia użytkowników, co pokrywa się z przyjętym w Glowing Waffle AMS podejściem opartym na kontroli dostępu. Dodatkowym atutem jest obsługa skanowania kodów kreskowych i kodów QR z poziomu aplikacji mobilnej, co może ułatwiać pracę w średnich i większych organizacjach.

Główne różnice między Asset Panda a Glowing Waffle AMS wynikają z modelu dostarczania oprogramowania. Asset Panda jest komercyjną platformą chmurową o zamkniętym kodzie, co oznacza zależność od zewnętrznego dostawcy w zakresie utrzymania systemu, dostępności usługi, przechowywania danych oraz dalszego rozwoju produktu. Jest to odmienne od założeń Glowing Waffle AMS, który przewiduje możliwość wdrożenia lokalnego, większą kontrolę nad danymi oraz ograniczenie ryzyka *vendor lock-in*.

Różnice dotyczą także kosztów i sposobu dostosowania systemu. Asset Panda działa w modelu subskrypcyjnym z wyceną indywidualną, a EZO Inventory publikuje cennik zależny między innymi od liczby środków trwałych. Wraz ze skalą użycia koszty takich rozwiązań mogą więc rosnąć. Ponadto elastyczność Asset Panda polega głównie na konfiguracji pól, formularzy i akcji, a nie na dedykowanym modelu dziedzicznym z hierarchią kategoria–typ, dziedziczeniem pól i przedrostkami tagów.

W rezultacie procesy takie jak wypożyczenia, incydenty czy kampanie inwentaryzacyjne w Asset Panda są odwzorowywane przez ogólne mechanizmy konfiguracji, natomiast w Glowing Waffle AMS stanowią osobne moduły dziedziczne. Asset Panda pozostaje jednym z bardziej elastycznych rozwiązań komercyjnych, ale jego chmurowy, zamknięty i subskrypcyjny charakter uzasadnia budowę systemu możliwego do wdrożenia lokalnego.

3. Propozycja systemu Glowing Waffle Asset Management System

Glowing Waffle AMS powstał z myślą o organizacjach potrzebujących centralnego narzędzia do zarządzania środkami trwałymi. Rozdział przedstawia propozycję systemu: jego wizję i założenia projektowe, słownik pojęć dziedzinowych, opis ról użytkowników oraz specyfikację wymagań funkcjonalnych i нефункциональных wraz z diagramami przypadków użycia.

3.1. Wizja systemu

Glowing Waffle Asset Management System (w skrócie: Glowing Waffle AMS) jest systemem informatycznym przeznaczonym do ewidencji, zarządzania i inwentaryzacji środków trwałych w organizacjach o złożonej strukturze. System adresowany jest do przedsiębiorstw o różnych skalach od małych (kilka – kilkanaście osób) po duże przedsiębiorstwa i instytucje – zarówno prywatne, jak i publiczne – które potrzebują kontroli nad posiadanymi zasobami, lecz nie wymagają pełnego pakietu ERP. Przykładowym odbiorcą systemu jest placówka oświatowa, potrzebująca zarządzania zasobami jednostki, ewidencją wyposażenia sal, sprzętu komputerowego i mienia ruchomego, z podziałem na lokalizacje i odpowiedzialnych za dany zasób pracowników.

Motywacją do stworzenia systemu jest obserwowana luka pomiędzy prostymi narzędziami inwentaryzacyjnymi a wysoce zaawansowanymi systemami *ERP*. Uproszczone narzędzia cechują się ograniczoną możliwością modelowania dziedziny – brak hierarchii kategorii, brak pól opisowych definiowanych przez użytkownika, brak szerszego zakresu obsługi – jak obsługa wypożyczeń, incydentów czy powiadomień. Systemy ERP zaś cechują się wysokim kosztem wdrożenia oraz utrzymania, są to zaawansowane i skomplikowane systemy, często surowe i mało przyjemne do obsługi, ze sztywną strukturą danych oraz niską elastycznością w dostosowaniu do specyfiki organizacji. Glowing Waffle AMS zajmuje przestrzeń pomiędzy obydwojema wariantami systemów, oferuje elastyczny model danych przy zachowaniu niskiego progu wdrożenia systemowego jak i pracownika.

Centralnym elementem systemu jest zasób – obiekt reprezentujący pojedynczy środek materialny organizacji. Administrator systemu może samodzielnie definiować kategorie i typy zasobów, a do każdej z tych klas przypisywać własne niestandardowe pola opisowe (np. numer seryjny, VIN, parametry techniczne). Każdy zasób posiada unikalny identyfikator – tag systemowy – oraz może zostać wygenerowany dla niego identyfikator zewnętrzny (np. kod zewnętrzny lub kod *QR*), co ułatwia pracę inwentaryzacyjną czy odczytanie informacji o środku trwałym z poziomu telefonu, przy pomocy przeglądarki internetowej oraz aparatu.

3.2. Założenia

System w swoim założeniu zaprojektowany jest z myślą o wdrożeniu lokalnym, z integracją z istniejącą infrastrukturą organizacji (w prototypie przedstawiona jest integracja z LDAP). System nie definiuje twardego limitu użytkowników ani środków. Dzięki elastycznej architekturze mikroservisowej możliwe jest niezależne, poziome skalowanie poszczególnych serwisów oraz stopniowe rozszerzanie i modyfikowanie systemu bez przerywania ciągłości działania pozostałych komponentów. Dzięki tej architekturze możliwe jest proste wprowadzanie nowych oraz usuwanie niepotrzebnych modułów na potrzeby indywidualne danej jednostki.

Głównym i na ten czas jedynym kanałem dostępu do systemu jest aplikacja webowa. Skanowanie identyfikatorów środków (kodów kreskowych, kodów QR) możliwe jest przy pomocy kamery urządzenia mobilnego w przeglądarce – bez potrzeby instalowania dedykowanej aplikacji mobilnej.

3.3. Słownik pojęć

AMS – (*Asset Management System* – system zarządzania zasobami) – system informatyczny odpowiedzialny za monitorowanie i zarządzanie zasobami firmy – fizycznych i cyfrowych. Umożliwia śledzenie cyklu życia środków w tym: lokalizacji, stanu technicznego, osób odpowiedzialnych za przypisane im zasoby. Zwiększa efektywność pracy i bezpieczeństwo operacyjne.

Dziennik zdarzeń – chronologiczny, niezmienny rejestr wszystkich operacji wykonanych na zasobie (m.in. edycji danych zasobu, wypożyczenia, incydenty, kampanie inwentaryzacyjne i rozbieżności w nich związane z tym zasobem). Prowadzony w trybie wyłącznie dopisywanym (ang. *append-only*) – zapewniającym niezmiennosc danych.

ERP – System ERP (*Enterprise Resource Planning* – Planowanie Zasobów Przedsiębiorstwa) to zintegrowane środowisko służące wieloma procesami firmy m.in. księgowość, kadry, inwentarz.

Identyfikator zewnętrzny zasobu – dodatkowy identyfikator powiązany z zasobem (tj. kod kreskowy, kod QR) generowany na podstawie UUID danego środka trwałego zapisanego w bazie. Umożliwia szybkie zidentyfikowanie zasobu poprzez np. skanowanie aparatem urządzenia mobilnego.

Incydent – zgłoszenie wynikłego problemu z zasobem (np. uszkodzenia, awaria, kradzież), z możliwością dołączenia plików dokumentujących zdarzenie (np. zdjęcia uszkodzenia) i opisu związanego z incydentem. Proces obejmuje wysłanie incydentu przez użytkownika i weryfikację przez pracownika z uprawnieniami do rozpatrywania tychże spraw.

Kampania inwentaryzacyjna – zaplanowany i zorganizowany proces weryfikacji faktycznego stanu zasobów firmy. Obejmuje skanowanie, weryfikację i wyznaczenie rozbieżności z przypisanym, a faktycznym stanem w trakcie realizowania inwentaryzacji.

Kategoria zasobu – grupa zasobów o wspólnym charakterze (np. „meble biurowe”, „sprzęt komputerowy”). Definiuje przedrostek (ang. *prefix*) tagu oraz zestaw pól niestandardowych dla przypisanych do kategorii zasobów.

Lokalizacja – fizyczne miejsce przechowywania środka trwałego, opisane trójpoziomą hierarchią: budynek, piętro, pomieszczenie.

Pola niestandardowe – zdefiniowane przez administratora pola opisowe, przypisywane do kategorii i typu zasobu (np. numer seryjny, VIN, wymiary).

Stan zasobu – aktualna kondycja zasobu (np. dobry, uszkodzony, zutylizowany).

Tag systemowy zasobu – unikalny identyfikator tekstowy nadawany automatycznie na podstawie przedrostka zdefiniowanego na poziomie kategorii oraz typu przypisanych do zasobu.

Typ zasobu – opcjonalne uszczegółowienie kategorii (np. „biurko” w „meble biurowe”, „laptop” w „sprzęt komputerowy”). Może definiować dodatkowe pola niestandardowe. Przypisane do typu zasoby dziedziczą po nim przedrostek, zamiast po kategorii.

Wypożyczenie – odnotowanie przekazania zasobu konkretnemu użytkownikowi systemu, wraz z datą wypożyczenia i planowaną datą jego zwrotu. Cały proces obejmuje wytyczenie prośby o wypożyczenie zasobu, akceptowanie lub odrzucenie prośby, monitoring czasu do końca wypożyczenia, powiadomienie w przypadku przekroczenia czasu wypożyczenia.

Zasób – zasób materialny, środek trwały będący w posiadaniu organizacji (np. laptop, drukarka, mebel biurowy, małe i duże AGD).

3.4. Role użytkowników i model uprawnień

System Glowing Waffle AMS rozróżnia użytkowników na podstawie ról przypisanych im w zewnętrznym serwerze tożsamości. Tożsamości pracowników oraz ich przynależność do grup pochodzą z firmowego katalogu LDAP, który stanowi źródło prawdy o zatrudnieniu i strukturze organizacyjnej. Keycloak pełni rolę dostawcy tożsamości – uwierzytelnia użytkownika i na podstawie danych z katalogu wystawia token dostępowy zawierający zestaw przypisanych ról. Sam mechanizm decydowania o tym, do jakich funkcji użytkownik ma dostęp, opiera się na modelu kontroli dostępu opartym na rolach (ang. *Role-Based Access Controll*, RBAC). Oznacza to, że uprawnienia nie są przypisywane bezpośrednio poszczególnym osobom, lecz całym rolom, a osoba uzyskuje dostęp do funkcji poprzez przynależność do roli, która ten dostęp posiada.

Model ról został zaprojektowany zgodnie z zasadą minimalnych uprawnień – każda rola obejmuje wyłącznie funkcje, które są niezbędne do realizacji powierzonych zadań. Dla każdej z nich przypisany jest również dostęp do funkcjonalności personalnych każdego pracownika. Dzięki czemu pracownik może przeglądać i obsługiwać wnioski o wypożyczenie sprzętu, zgłaszać problemy z nimi związane oraz wyświetlać przypisane do niego zasoby – wszystkie te dostępy zwężone są do właściciela, tzn. pracownik nie ma dostępu do informacji innych pracowników.

W systemie wyróżniono następujące role.

- **Pracownik** (ams_user) – stanowi podstawową rolę przypisywaną każdemu zalogowanemu użytkownikowi systemu. Pełni ona uzupełnienie codziennej pracy w organizacji – pracownik może składać wnioski o wypożyczenie środków trwałych, zgłaszać incydenty oraz przeglądać i obsługiwać wyłącznie własne rekordy, takie jak złożone przez siebie wnioski czy zarejestrowane zgłoszenia. Dostęp do danych innych osób pozostaje dla tej roli niedostępny, nie ma również możliwość edycji wypożyczonych przez niego zasobów.
- **Zarządca środków** (asset_manager) – odpowiada za zarządzanie ewidencją środków trwałych w systemie. Do jego zadań należy wprowadzanie i aktualizowanie danych o środkach trwałych, prowadzenie finansowej ewidencji środków, w tym stawek VAT i danych zakupowych, zarządzanie lokalizacjami, kategoriami oraz definicjami pól opisujących środki.
- **Audytory inwentaryzacji** (inventory_auditor) – zarządza procesem przeprowadzania kampanii inwentaryzacyjnych. Pracownik z przypisaną tą rolą ma możliwość tworzenia i zarządzania kampaniami oraz rozpoczęcie i ich zakończenie.
- **Inwentaryzator** (inventory_counter) – uczestniczy w procesie inwentaryzacji. Jego rola polega na fizycznym sprawdzeniu środków przypisanych do inwentaryzacji i ich zeskanowaniu lub ręcznego odznaczenia w systemie.
- **Zarządca wypożyczeń** (loan_approver) – odpowiada za obsługę procesu wypożyczeń środków. Odpowiedzialny jest za zatwierdzanie lub odrzucanie wniosków o wypożyczenie złożonych przez pracowników oraz obsługa przekazania środków między użytkownikami.
- **Obsługujący incydenty** (incident_handler) – obsługuje zgłoszenia dotyczących środków trwałych – przegląda, prowadzi i rozwiązuje.
- **Administrator systemu** (ams_admin) – posiada pełnię praw w systemie, niezależnie od podziału kompetencji pomiędzy pozostałe role. Rola ta odpowiada za nadzór nad całością działania systemu.

Przedstawiony podział ról umożliwia odwzorowanie typowej struktury kompetencyjnej przedsiębiorstwa, w której zadania odrębne kategorie zadań – inwentaryzacja, wypożyczenia, itd. są rozdzielone na poszczególne grupy pracowników.

3.5. Wymagania funkcjonalne

Podrozdział przedstawia wymagania funkcjonalne pogrupowane według obszarów dziedzinowych systemu. Tabela 3–11 zestawiają operacje wraz z opisem, aktorami oraz warunkami początkowym – dla wypożyczeń i kampanii wyodrębniono działania realizowane automatycznie przez system.

Tabela 3 Wymagania funkcjonalne - zarządzanie zasobami. Źródło: Opracowanie własne.

Lp.	Nazwa	Opis	Aktorzy	Warunki początkowe
1	Dodaj zasób	Wprowadzenie nowego środka trwałego wraz z kategorią, typem, lokalizacją, stawką VAT, dokumentem zakupu, stanem i osobą odpowiedzialną.	Zarządca środków	W systemie zdefiniowano uprzednio kategorię, lokalizację, dokument zakupu.
2	Edytuj zasób	Modyfikacja danych istniejącego środka trwałego	Zarządca środków	Zasób nie jest zablokowany przez trwającą inwentaryzację.
3	Usuń zasób	Trwałe usunięcie zasobu	Zarządca środków	Zasób nie jest zablokowany przez trwającą inwentaryzację.
4	Przeglądaj katalog zasobów	Wyświetlenie listy zasobów z możliwością wyszukiwania i filtrowania.	Zarządca środków	
5	Wyświetl szczegóły zasobu	Podgląd atrybutów zasobu wraz z wartościami pól niestandardowych i identyfikatorami.	Zarządca środków	
6	Wyświetl moje zasoby	Lista zasobów, dla których zalogowany użytkownik jest osobą odpowiedzialną.	Wszyscy użytkownicy	
7	Wyświetl historię zasobu	Odczyt chronologicznej osi zdarzeń (ang. timeline) zasobu.	Zarządca środków,	Zasób posiada co najmniej jeden wpis w dzienniku zdarzeń.

Tabela 4 Wymagania funkcjonalne - klasyfikacja zasobów i pola niestandardowe.
Źródło: Opracowanie własne.

Lp.	Nazwa	Opis	Aktorzy	Warunki początkowe
1	Dodanie kategorii zasobu	Utworzenie kategorii wraz z unikalnym przedrostkiem tagu.	Zarządca środków	Przedrostek tagu nie jest zajęty przez żadną kategorię ani typ.
2	Edycja kategorii zasobu	Zmiana danych kategorii oraz modyfikacja przypisanych do kategorii definicji pól.	Zarządca środków	Jeżeli zostaje zmieniany przedrostek musi on być unikalny.
3	Usunięcie kategorii zasobu	Usunięcie kategorii zasobu.	Zarządca środków	Kategoria nie ma przypisanych zasobów, typów ani definicji pól.
4	Dodanie typu zasobu	Utworzenie typu środka trwałego jako uszczegółowienia kategorii, z opcjonalnym własnym przedrostkiem i dodatkowymi definicjami pól niestandardowych.	Zarządca środków	Istnieje kategoria nadrzędna. Jeżeli podano przedrostek – musi być unikalny w obrębie kategorii i typów.
5	Edycja typu zasobu	Zmiana danych typu – kategorii nadrzędnej, przedrostka oraz przypisanych pól.	Zarządca środków	W przypadku zmiany lub przypisania przedrostku musi on zostać unikalny.
6	Usunięcie typu zasobu	Usunięcie typu zasobu.	Zarządca środków	Typ nie ma przypisanych zasobów.
7	Przeglądanie kategorii i ich szczegółów.	Wyświetlenie listy z dostępnymi kategoriami z możliwością odczytania szczegółów każdej z nich.	Zarządca środków	
8	Przeglądanie typów i ich szczegółów.	Wyświetlenie listy z dostępnymi typami z możliwością odczytania szczegółów każdego z nich.	Zarządca środków	
9	Tworzenie definicji pola niestandardowego	Utworzenie pola opisowego o określonym typie (tekst, liczba, data, wartość logiczna).	Zarządca środków	
10	Edycja definicji pola niestandardowego	Zmiana nazwy, etykiety lub typu pola.	Zarządca środków	
11	Usunięcie definicji pola niestandardowego	Usunięcie pola niestandardowego	Zarządca środków	Pole nie jest przypisane do żadnej kategorii ani typu.

Tabela 5 Wymagania funkcjonalne - zarządzanie lokalizacjami w strukturze organizacji.
Źródło: Opracowanie własne.

Lp.	Nazwa	Opis	Aktorzy	Warunki początkowe
1	Dodanie lokalizacji	Definiowanie lokalizacji w trzech poziomach: budynek, piętro, pomieszczenie.	Zarządca środków	
2	Edycja lokalizacji	Możliwość zmiany istniejącej hierarchii lub jej usunięcie.	Zarządca środków	
	Usunięcie lokalizacji	Usunięcie lokalizacji	Zarządca środków	Lokalizacja nie ma przypisanych zasobów
3	Przeglądanie lokalizacji i ich szczegółów.	Wylistowanie dostępnych lokalizacji wraz z dostępem do szczegółowych informacji – przede wszystkim przypisanymi do nich zasobami.	Zarządca środków	

Tabela 6 Wymagania funkcjonalne - finansowa ewidencja zasobów. Źródło: Opracowanie własne.

Lp.	Nazwa	Opis	Aktorzy	Warunki początkowe
1	Dodanie stawki VAT	Utworzenie stawki VAT, nowa stawka jest automatycznie aktywna.	Zarządca środków	Nazwa stawki nie jest zajęta.
2	Edycja / dezaktywacja stawki VAT	Zmiana danych stawki, w tym jej dezaktywacja.	Zarządca środków	
3	Usunięcie stawki VAT	Trwałe usunięcie stawki VAT	Zarządca środków	
4	Przeglądanie stawek VAT i ich szczegółów	Wylistowanie stawek VAT i dostęp do szczegółów dla każdej z nich.	Zarządca środków	
5	Dodanie dokumentu zakupowego	Rejestracja faktury wraz z sumami netto i brutto oraz walutą.	Zarządca środków	Suma brutto nie może być mniejsza niż netto.
6	Edycja dokumentu zakupowego	Modyfikacja danych dokumentu zakupowego.	Zarządca środków	Suma brutto nie może być mniejsza niż netto.
7	Usunięcie dokumentu zakupowego	Usunięcie dokumentu zakupowego.	Zarządca środków	Dokument nie ma przypisanych zasobów.
8	Przeglądanie dokumentów i ich szczegółów	Wyświetlanie dostępnych dokumentów wraz z dostępem do szczegółów dla każdej z nich.	Zarządca środków	

Tabela 7 Wymagania funkcjonalne - Wypożyczenia środków trwałych. Źródło: Opracowanie własne.

Lp.	Nazwa	Opis	Aktorzy	Warunki początkowe
1	Złożenie wniosku o wypożyczenie	Utworzenie wniosku na jeden lub wiele zasobów z uzasadnieniem i okresem.	Wszyscy pracownicy z dostępem do systemu	Zasoby nie są zablokowane inwentaryzacją i są dostępne w żądanym terminem.
2	Edycja wniosku	Możliwość zmiany terminu i zakresu wypożyczanych zasobów dla wypożyczającego.	Zarządca wypożyczeń	
3	Przeglądanie procesów związanych z wnioskami	Możliwość wylistowania wniosków o wypożyczenie, decyzji, przekazania wypożyczeń oraz zakończonych wypożyczeń	Zarządca wypożyczeń	
4	Rozpatrzenie wniosku	Akceptacja bądź odrzucenie wniosku wraz komentarzem.	Zarządca wypożyczeń	Wniosek ma status oczekujący. Podczas zatwierdzania wniosku pracownik nie posiada nierozliczonych przeterminowań, a sam zasób jest dostępny.
5	Wydanie zasobu przed czasem	Ręczne wydanie zaplanowanego wypożyczenia przed ustalonym okresem.	Zarządca wypożyczeń	Wypożyczenie ma status jako zaplanowane, a zasób jest dostępny.
6	Zarejestrowanie zwrotu	Odniesienie zwrotu środka trwałego.	Zarządca wypożyczeń	Wypożyczenie ma status jako aktywne lub przeterminowane.
7	Przekazanie wypożyczenia	Przekazanie aktywnego wypożyczenia innemu użytkownikowi.	Zarządca wypożyczeń	Wypożyczenie jest aktywne.
8	Przeglądanie osobistych wypożyczeń	Lista wypożyczeń powiązanych z wnioskami użytkownika.	Wszyscy pracownicy z dostępem do systemu	
9	Przeglądanie osobistych wniosków	List wniosków złożonych przez użytkownika.	Wszyscy pracownicy z dostępem do systemu	

Tabela 8 Wymagania funkcjonalne - wypożyczenia (działania automatyczne systemu). Źródło: Opracowanie własne.

Lp.	Nazwa	Opis	Aktorzy	Warunki początkowe
1	Automatyczna aktywacja wypożyczenia	Przejsie zaplanowanego wypożyczenia w stan aktywny po nadejściu terminu rozpoczęcia.	System	Nadszedł termin rozpoczęcia, a zasób jest fizycznie wolny (cykl co 5 min).
2	Oznaczenie przeterminowania	Zmiana statusu wypożyczenia na przeterminowane po przekroczeniu terminu zwrotu.	System	Wypożyczenie aktywne lub zaplanowane przekroczyło termin zwrotu (cykl co 5 min).
3	Przypomnienie o zbliżającym się zwrocie	Wysłanie powiadomienia e-mail o zbliżającym się terminie zwrotu.	System	Termin zwrotu w ciągu ok. 7 dni (przypomnienie wysyłane jednokrotnie).
4	Powiadomienie o przeterminowaniu	Wysłanie powiadomienia e-mail o przekroczeniu terminu zwrotu.	System	Wypożyczenie stało się przeterminowane (powiadomienie jednokrotne).
5	Zapis zdarzeń wypożyczenia w historii zasobu	Dopisanie zdarzeń wypożyczenia (aktywacja, zwrot, przeterminowanie) do historii danego zasobu.	System	Nastąpiła zmiana stanu wypożyczenia (aktywacja, zwrot lub przeterminowanie)

Tabela 9 Wymagania funkcjonalne - kampanie inwentaryzacyjne. Źródło: Opracowanie własne.

Lp.	Nazwa	Opis	Aktorzy	Warunki początkowe
1	Utworzenie kampanii	Zaplanowanie kampanii obejmującej wskazane środki trwałe lub całe lokalizacje wraz z przypisaniem inwentaryzatorów i daty rozpoczęcia inwentaryzacji	Audytory inwentaryzacji	Kod kampanii jest unikalny. Wskazano co najmniej jedną lokalizację lub zasób oraz co najmniej jednego inwentaryzatora.
2	Modyfikacja kampanii	Modyfikacja danych i zakresu kampanii.	Audytory inwentaryzacji	Kampania posiada status utworzonej - po wystartowaniu kampanii i w kolejnych stanach modyfikacja nie jest możliwa.
3	Uruchomienie kampanii	Rozpoczęcie kampanii inwentaryzacyjnej.	Audytory inwentaryzacji	Data rozpoczęcia nie jest w przyszłości (chyba że wymuszono start), a ewentualna data zakończenia nie jest przeszła.

4	Zakończenie kampanii	Zamknięcie etapu skanowania i przejście do podsumowania wyników.	Audytor inwentaryzacji	Kampania jest aktywna.
5	Skanowanie zasobu	Potwierdzenie obecności zasobu przez skan identyfikatora/tagu lub odznaczenie ręczne, wraz z lokalizacją i obserwowanym stanem.	Audytor inwentaryzacji, Inwentaryzator	Kampania ma status aktywna; podano lokalizację oraz identyfikator lub tag zasobu
6	Synchronizacja stanu kampanii	Ręczna możliwość odświeżenia stanu kampanii w celu przeliczenia i pobrania przypisanych zasobów w przypadku zmian w lokalizacjach i zasobach (w głównych punktach przebiegu inwentaryzacji system robi to automatycznie)	Audytor inwentaryzacji, Inwentaryzator	
7	Przeglądanie przypisanych kampanii	Lista kampanii, do których przypisany jest zalogowany użytkownik.	Audytor inwentaryzacji, Inwentaryzator	
8	Przeglądanie wszystkich kampanii	Lista wszystkich kampanii w organizacji.	Audytor inwentaryzacji	
9	Przeglądanie szczegółów kampanii	Podgląd kampanii wraz z lokalizacjami, zasobami i postępem skanowania.	Audytor inwentaryzacji, Inwentaryzator	
10	Przeglądanie rozbieżności	Wgląd w rozbieżności wykryte w ramach kampanii.	Audytor inwentaryzacji, Inwentaryzator	
11	Przeglądanie dziennika operacji	Wgląd w chronologiczny dziennik operacji kampanii.	Audytor inwentaryzacji, Inwentaryzator	

Tabela 10 Wymagania funkcjonalne – kampanie inwentaryzacyjne (działania automatyczne systemu).
Źródło: opracowanie własne.

Lp.	Nazwa	Opis	Aktorzy	Warunki początkowe
1	Zablokowanie zasobów na czas inwentaryzacji	Objęte kampanią zasoby zostają zablokowane przed edycją, usunięciem i wypożyczeniem.	System	Kampania została uruchomiona
2	Odblokowanie zasobów i aktualizacja daty inwentaryzacji	Zwolnienie blokady zasobów oraz odnotowanie daty ostatniej inwentaryzacji.	System	Kampania została zamknięta

3	Rejestracja rozbieżności podczas skanowania	Automatyczne utworzenie rozbieżności: zła lokalizacja, niezgodność stanu lub zasób nadmiarowy.	System	Zeskanowano zasób niezgodny z oczekiwaniami lub spoza zakresu kampanii
4	Rejestracja braków po zamknięciu	Oznaczenie zasobów planowanych, lecz niezeskanowanych jako brakujące.	System	Trwa zamykanie kampanii
5	Zapis zakończenia inwentaryzacji w historii zasobów	Dopisanie zdarzeń podsumowujących inwentaryzację do historii każdego zasobu.	System	Kampania została zamknięta

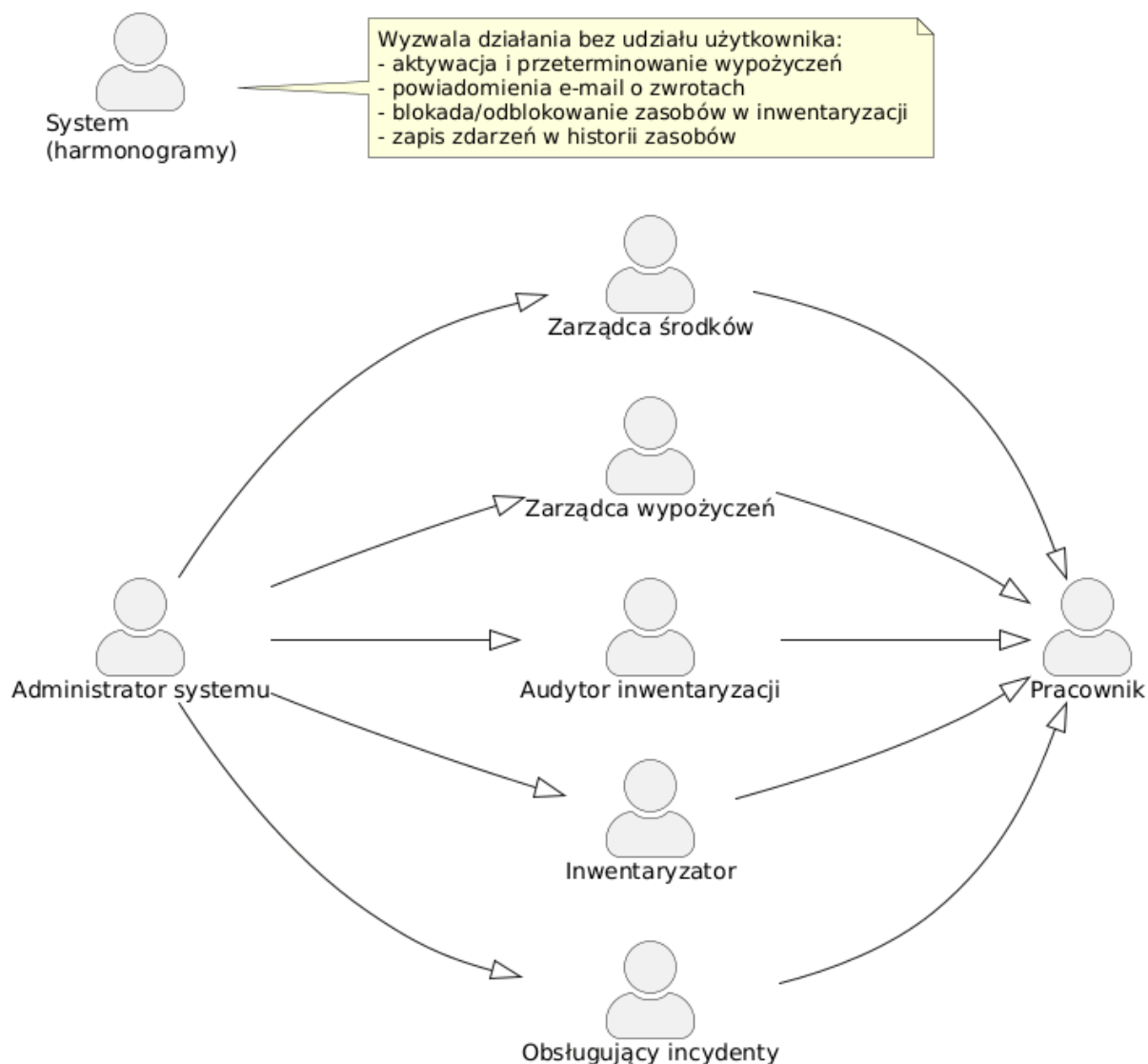
Tabela 11 Wymagania funkcjonalne - obsługa incydentów. Źródło: Opracowanie własne

Lp.	Nazwa	Opis	Aktorzy	Warunki początkowe
1	Zgłoszenie incyduentu wypożyczonego sprzętu	Utworzenie zgłoszenia z problemem wypożyczonego sprzętu	Wszyscy pracownicy z dostępem do systemu	Środek trwały jest przypisany do zgłaszającego
2	Aktualizacja incyduentu	Zmiana danych zgłoszenia: tytułu, priorytetu, statusu, przypisania lub zasobu.	Obsługujący incyduenty	
3	Wyświetlenie szczegółów incyduentu	Podgląd pojedynczego zgłoszenia.	Obsługujący incyduenty	
4	Przeglądanie wszystkich incydentów	Lista wszystkich zgłoszeń z możliwością filtrowania.	Obsługujący incyduenty	
5	Przeglądanie własnych incydentów	Lista zgłoszeń utworzonych przez użytkownika.	Wszyscy pracownicy z dostępem do systemu	
6	Usunięcie incyduentu	Trwałe usunięcie zgłoszenia wraz z jego załącznikami.	Administrator systemu	
7	Zamknięcie incyduentu	Zamknięcie zgłoszenia	Obsługujący incyduenty	
8	Dodanie komentarza	Dopisanie komentarza do zgłoszenia.	Wszyscy pracownicy z dostępem do systemu	Użytkownik jest przypisany do zgłoszenia.

9	Edycja komentarza	Modyfikacja treści komentarza.	Wszyscy pracownicy z dostępem do systemu, Obsługujący incydenty	Użytkownik jest przypisany do zgłoszenia i próbuje edytować swój komentarz.
10	Usunięcie komentarza	Usunięcie własnego komentarza ze zgłoszenia.	Wszyscy pracownicy z dostępem do systemu	Użytkownik jest przypisany do zgłoszenia i próbuje usunąć swój komentarz.
11	Dodanie załącznika	Dołączenie pliku dokumentującego zdarzenie (np. zdjęcia uszkodzenia).	Wszyscy pracownicy z dostępem do systemu, Obsługujący incydenty	Użytkownik jest przypisany do zgłoszenia.
12	Przeglądanie i pobieranie załączników	Podgląd listy załączników i pobranie pliku.	Wszyscy pracownicy z dostępem do systemu, Obsługujący incydenty	Użytkownik jest przypisany do zgłoszenia.
13	Usunięcie załącznika	Usunięcie załącznika ze zgłoszenia.	Obsługujący incydenty	

3.6. Aktorzy systemu

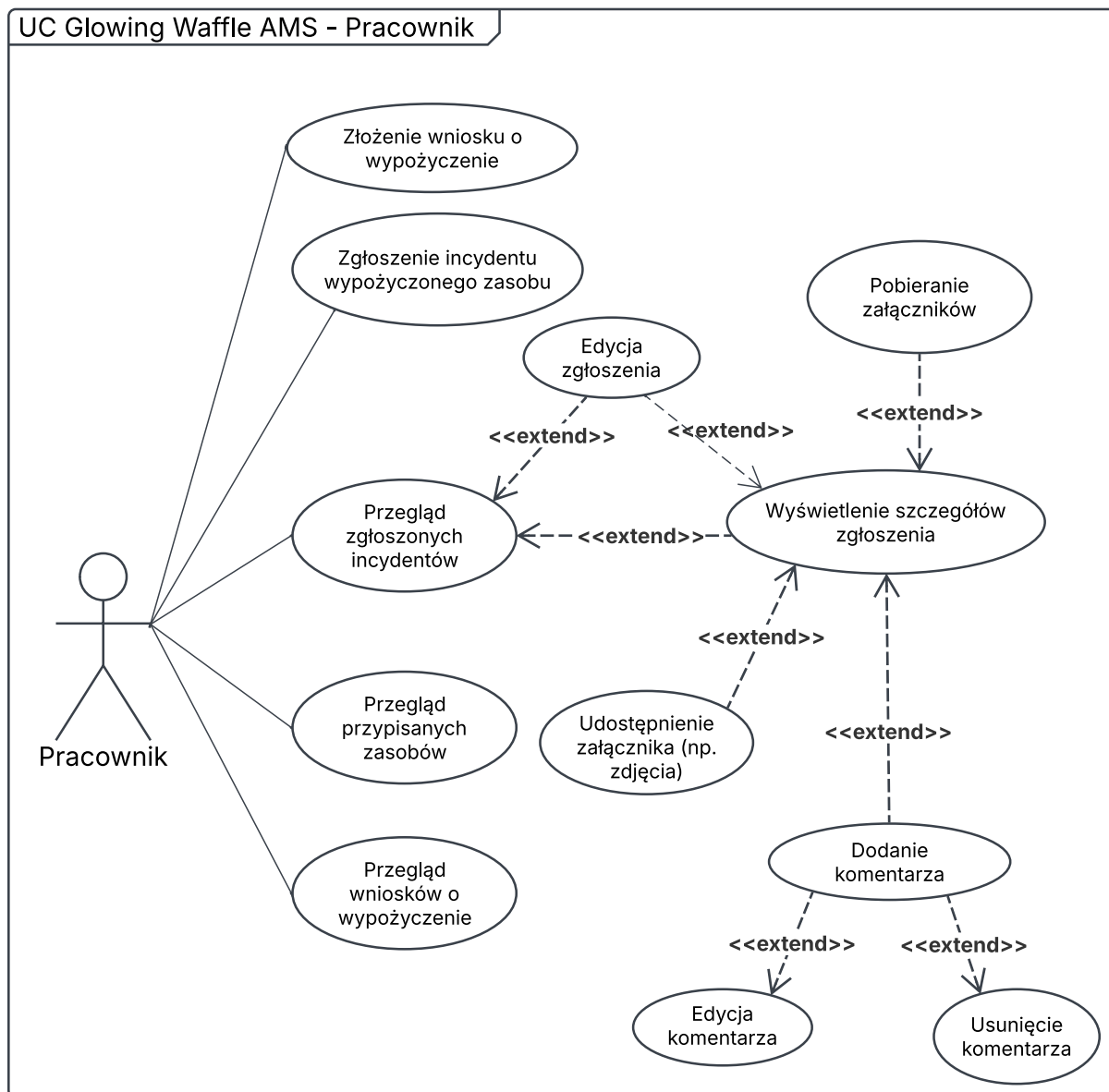
Diagram (Rysunek 9) przedstawia aktorów systemu wynikających z modelu ról: rolę bazową Pracownik, role funkcyjne oraz Administratora systemu. Relacja wyraża dziedziczenie funkcji Pracownika przez role funkcyjne, wyodrębniono również aktora System dla działań automatycznych.



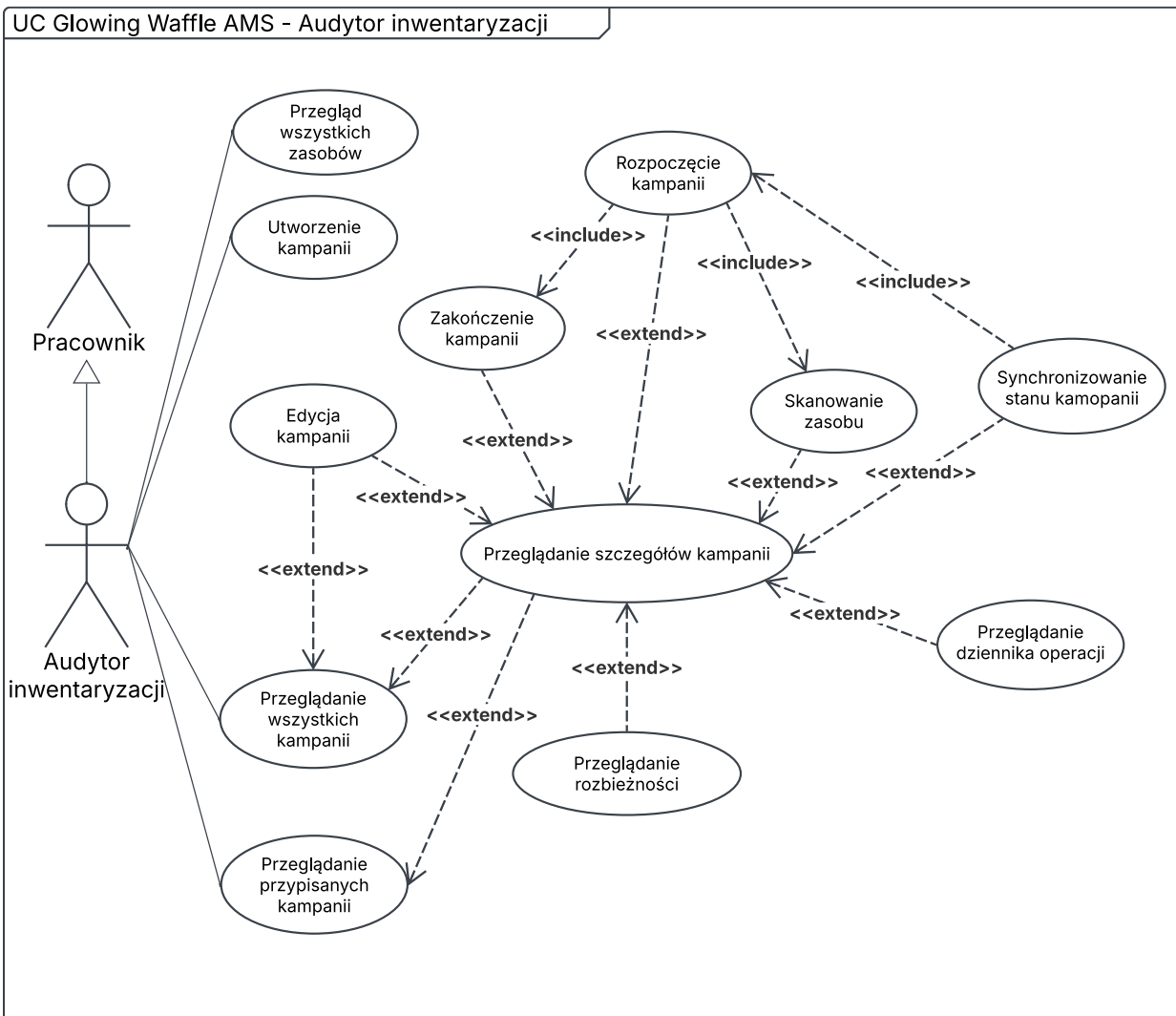
Rysunek 9 Diagram aktorów systemu Glowing Waffle AMS. Źródło: Opracowanie własne.

3.7. Przypadki użycia

Na podstawie wymagań z podrozdziału 3.5 opracowano diagramy przypadków użycia dla poszczególnych ról, z relacjami zawierania i rozszerzenia. Przypadki wspólne dla roli Pracownik są dziedziczone i nie są powielane na diagramach ról funkcyjnych.

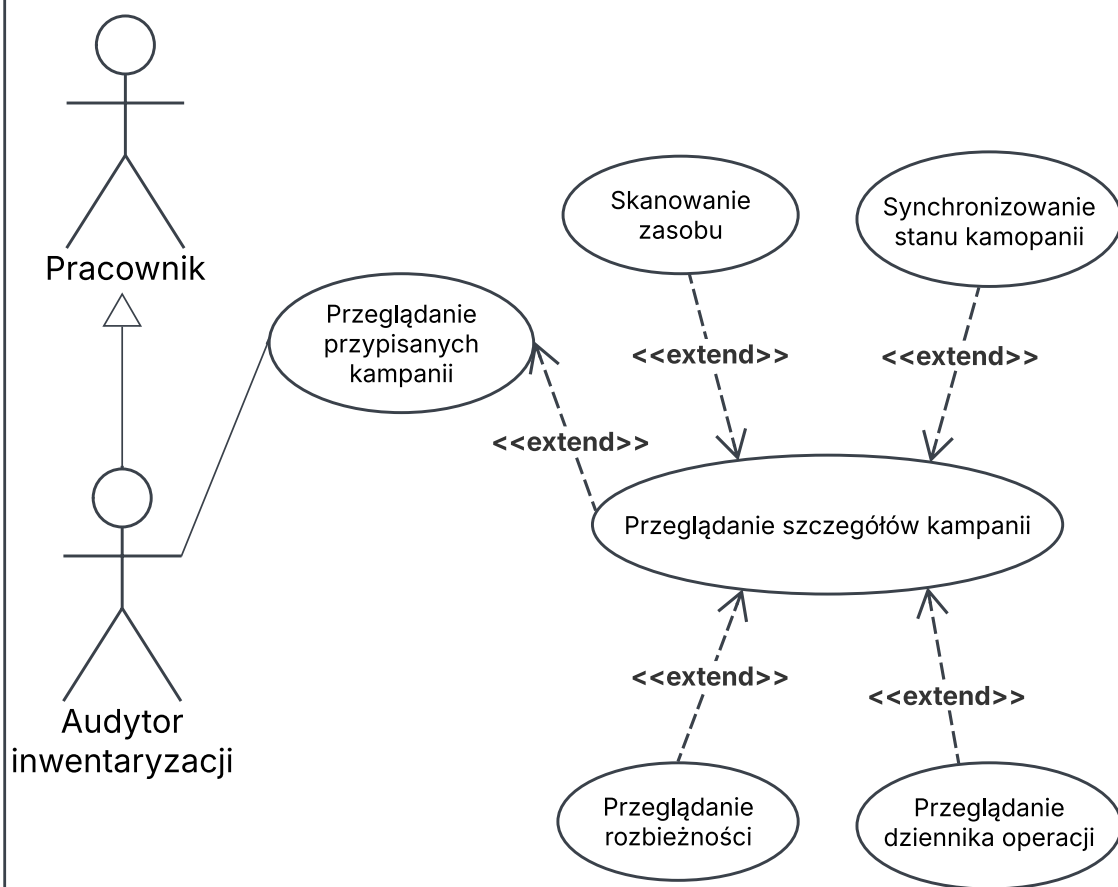


Rysunek 10 Diagram przypadków użycia dla roli Pracownik. Źródło: Opracowanie własne.

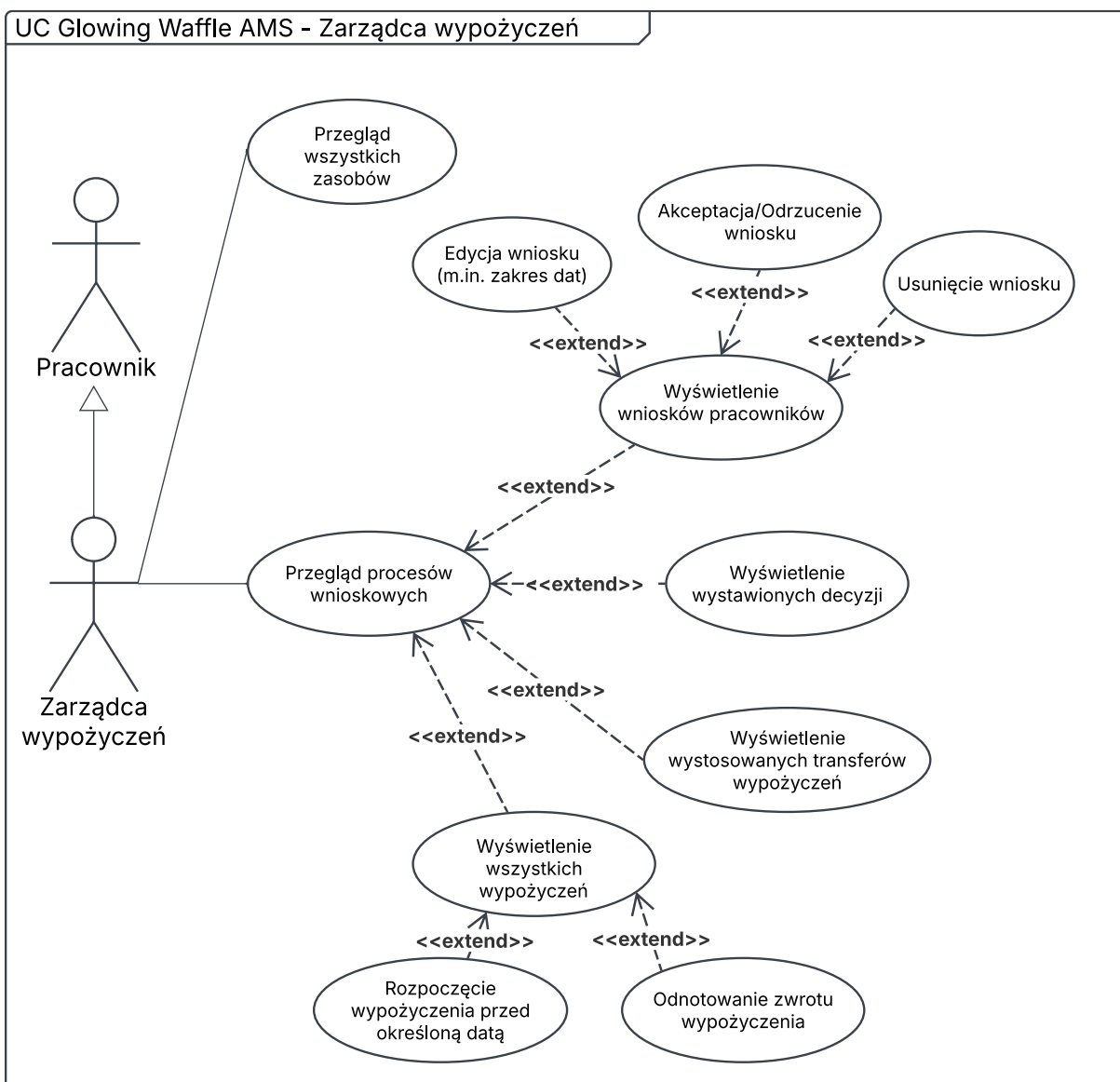


Rysunek 11 Diagram przypadków użycia dla roli Audytor inwentaryzacji. Źródło: Opracowanie własne.

UC Glowing Waffle AMS - Inwentaryzator

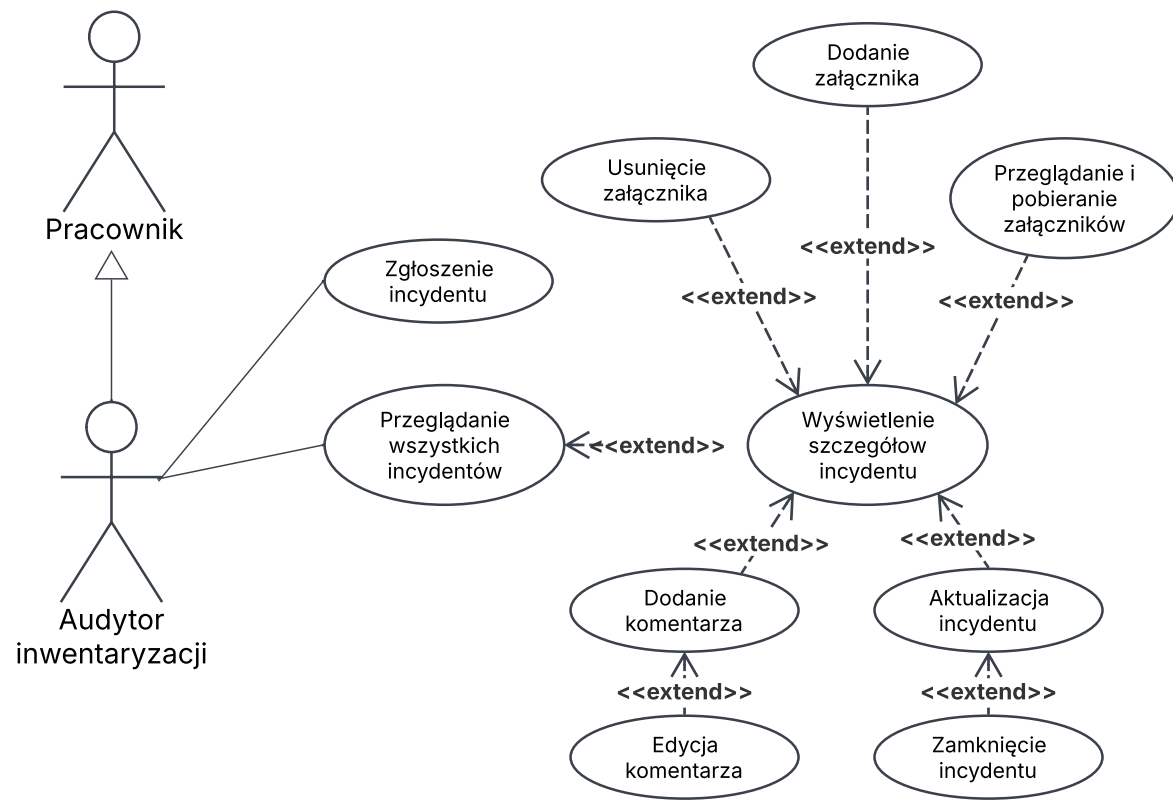


Rysunek 12 Diagram przypadków użycia dla roli Inwentaryzator. Źródło: Opracowanie własne.

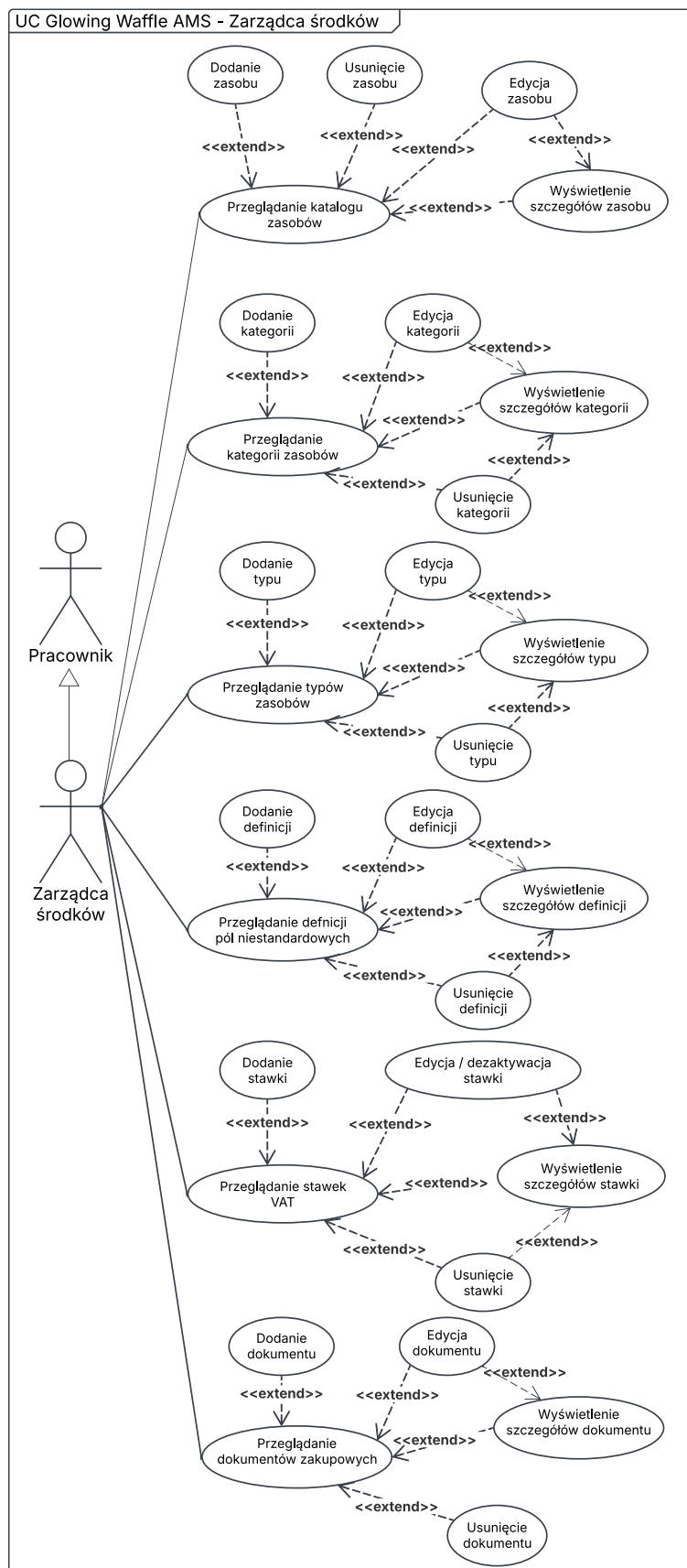


Rysunek 13 Diagram przypadków użycia dla roli Zarządca wypożyczeń. Źródło: Opracowanie własne.

UC Glowing Waffle AMS - Obsługujący incydenty



Rysunek 14 Diagram przypadków użycia dla roli Obsługujący incydenty. Źródło: Opracowanie własne.



Rysunek 15 Diagram przypadków użycia dla roli Zarządca środków. Źródło: Opracowanie własne.

3.8. Wymagania niefunkcjonalne

Tabela 12 prezentuje wymagania niefunkcjonalne stawiane systemowi Glowing Waffle AMS, w podziale na cel oraz wskaźnik; branżowo, cel określany jest często angielskim wyrażeniem *Service Level Objective*, a zaś wskaźnik *Service Level Indicator* [8].

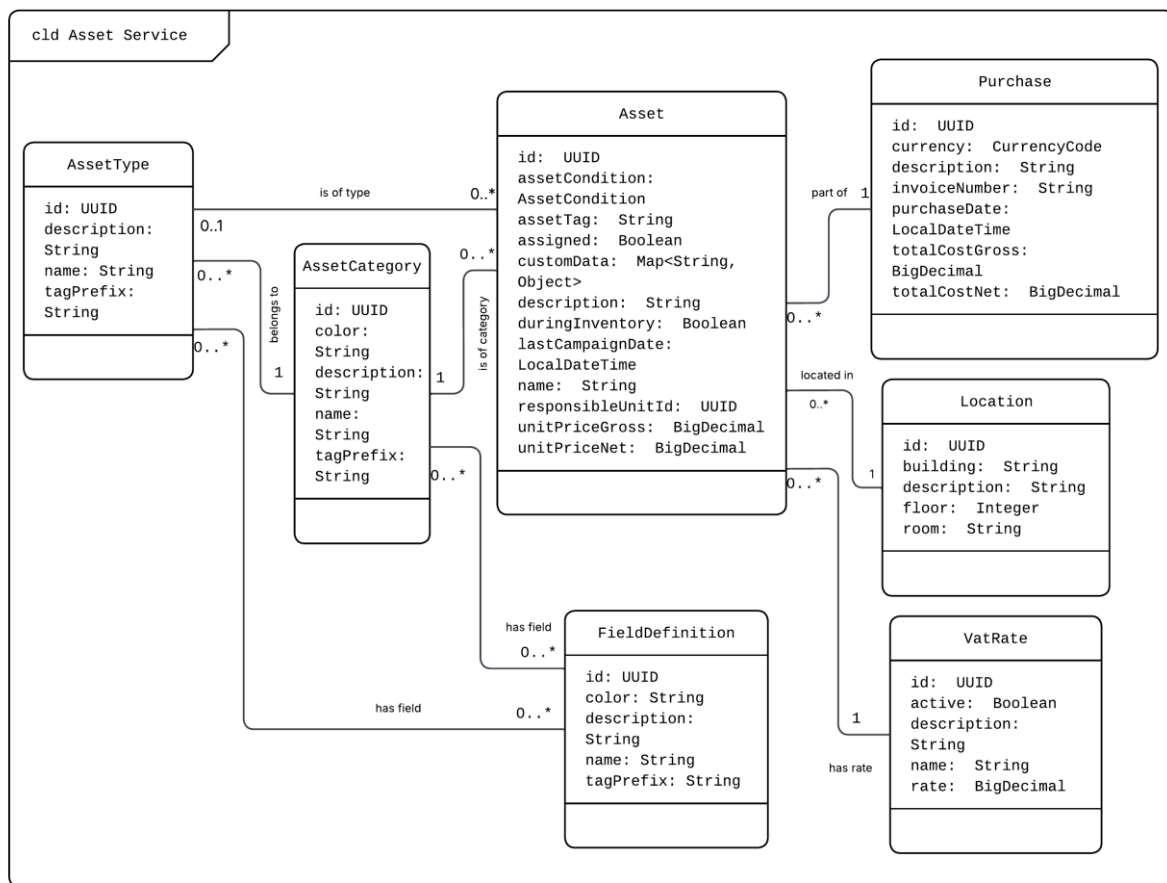
Tabela 12 SLO oraz SLI systemu Glowing Waffle AMS. Źródło: Opracowanie własne.

<i>Service Level Objective</i> (cel)	<i>Service Level Indicator</i> (wskaźnik)
Żądania GET są obsługiwane w czasie poniżej 700 milisekund dla 99% żądań przy założeniu 100 współbieżnych użytkowników.	Opóźnienie po stronie serwera, mierzone w bramie API: $P90 \leq 300$ milisekund, $P99 \leq 700$ milisekund
Żądania POST są obsługiwane w czasie poniżej 900 milisekund dla 99% żądań przy założeniu 100 współbieżnych użytkowników.	Opóźnienie po stronie serwera, mierzone w bramie API: $P90 \leq 400$ milisekund, $P99 \leq 900$ milisekund
Żądania PUT są obsługiwane w czasie poniżej 900 milisekund dla 99% żądań przy założeniu 100 współbieżnych użytkowników.	Opóźnienie po stronie serwera, mierzone w bramie API: $P90 \leq 400$ milisekund, $P99 \leq 900$ milisekund
Żądania DELETE są obsługiwane w czasie poniżej 800 milisekund dla 99% żądań przy założeniu 100 współbieżnych użytkowników.	Opóźnienie po stronie serwera, mierzone w bramie API: $P90 \leq 350$ milisekund, $P99 \leq 800$ milisekund

3.9. Projektowy diagram klas

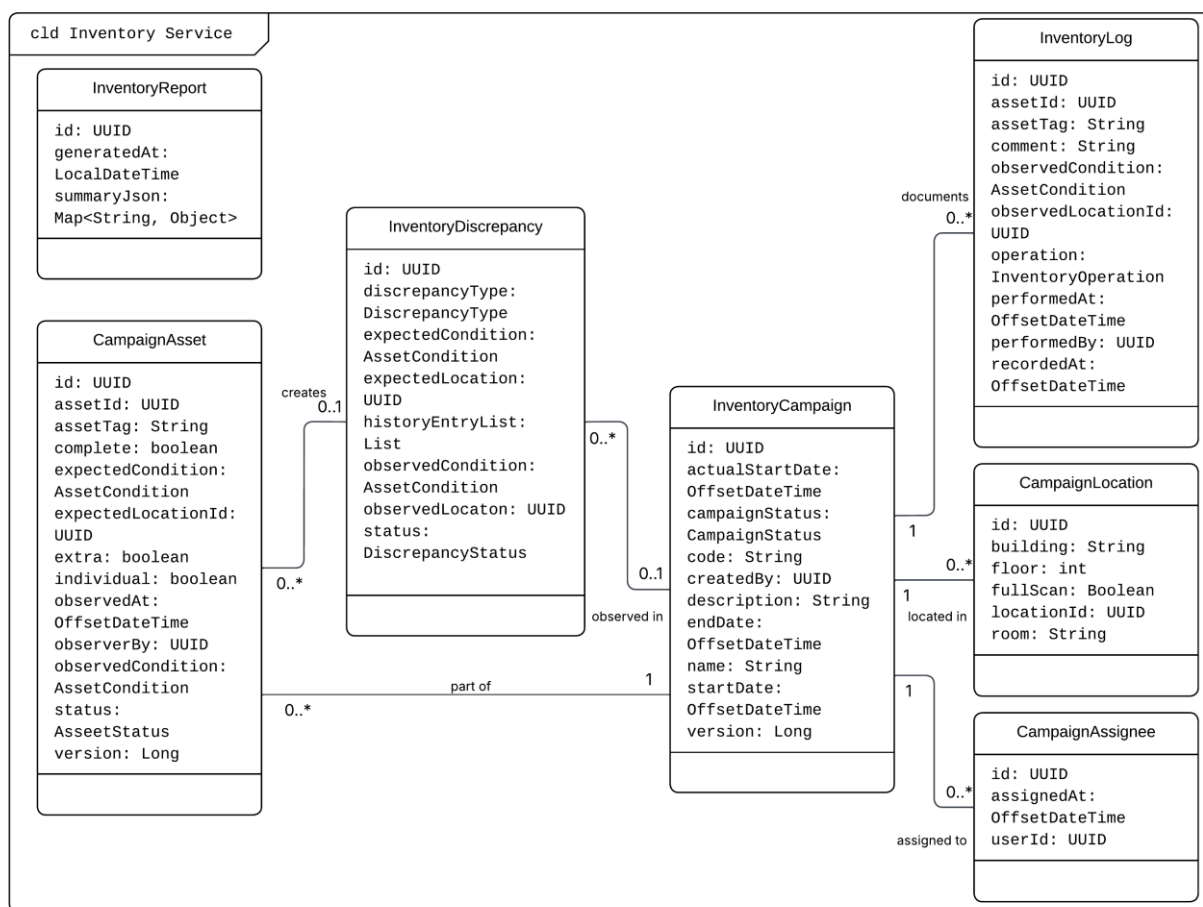
Diagramy przedstawiają model klas dziedzicznych systemu wraz z ich atrybutami i powiązaniami.

3.9.1 Asset Service



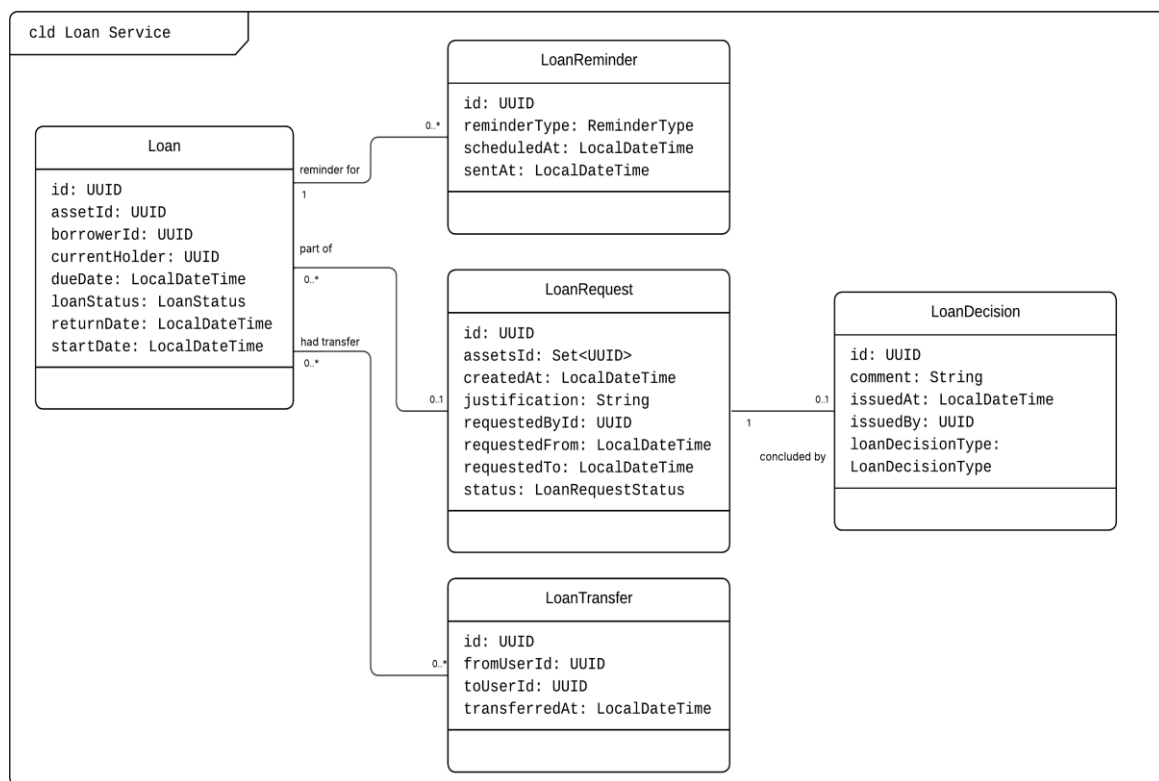
Rysunek 16 Diagram klas mikroserwisu Asset Service. Źródło: Opracowanie własne.

3.9.2 Inventory Service



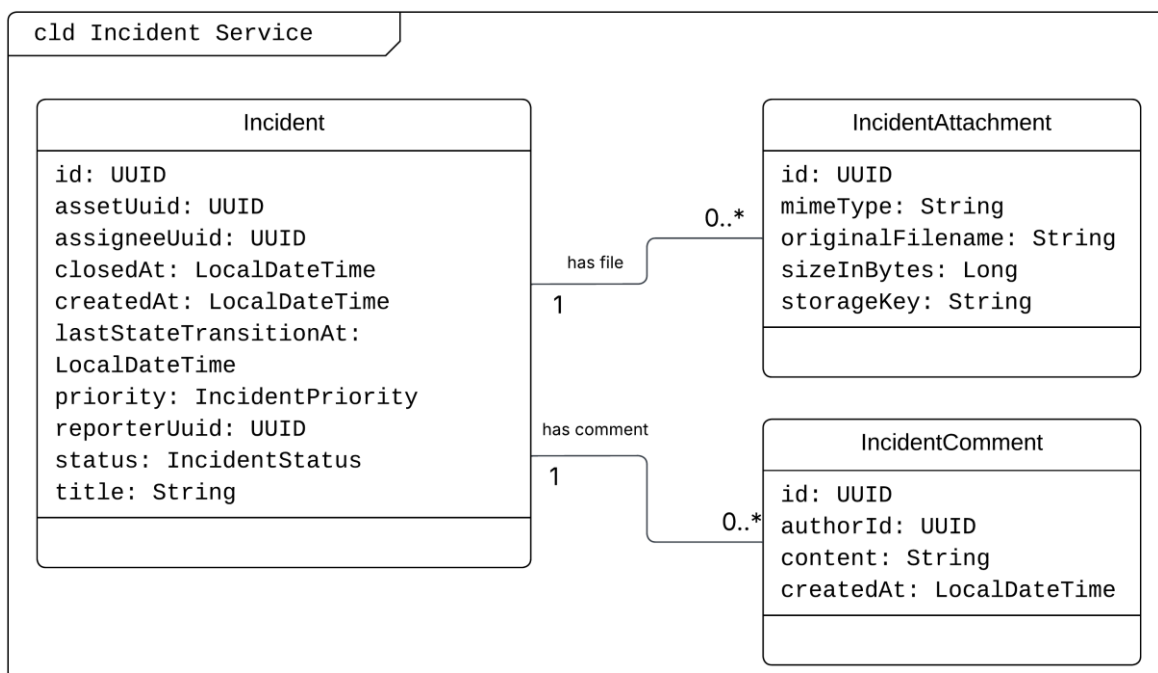
Rysunek 17 Diagram klas mikroservisu Inventory Service. Źródło: Opracowanie własne.

3.9.3 Loan Service



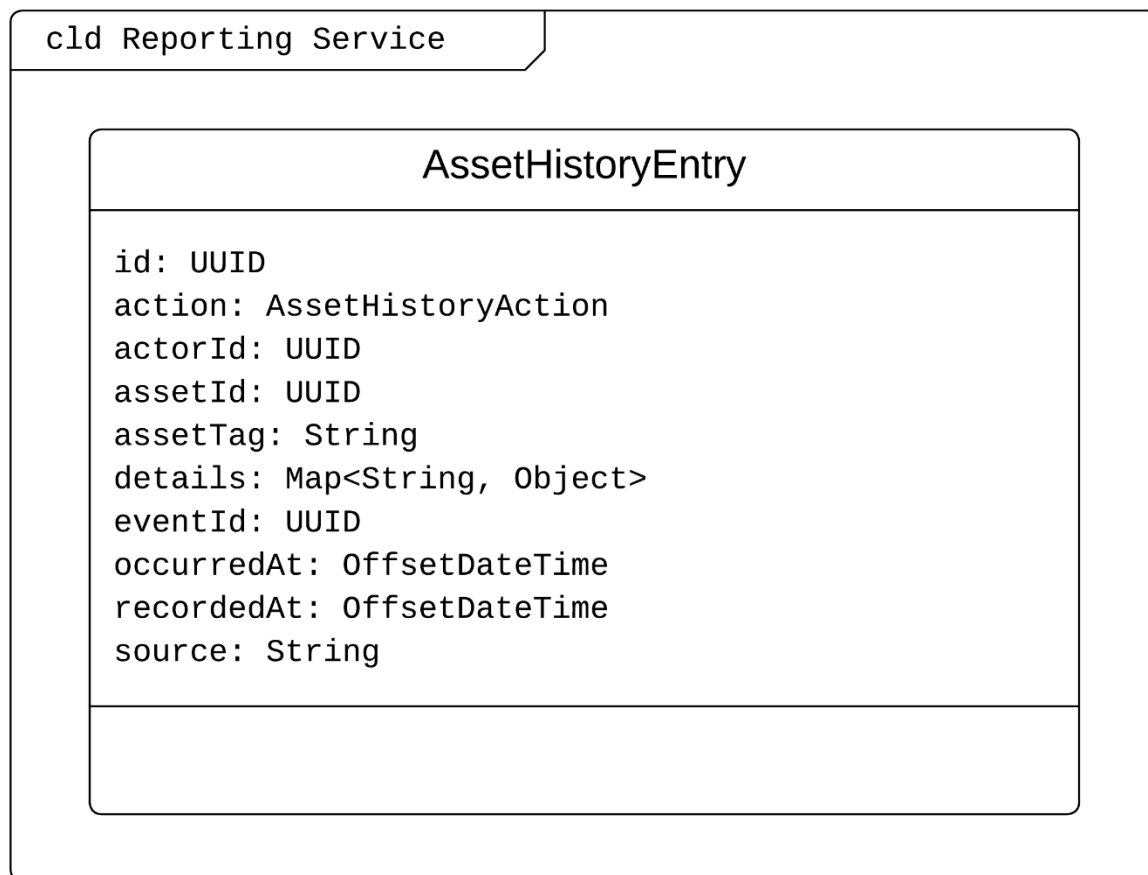
Rysunek 18 Diagram klas mikroserwisu Loan Service. Źródło: Opracowanie własne.

3.9.4 Incident Service



Rysunek 19 Diagram klas mikroserwisu Incident Service. Źródło: Opracowanie własne

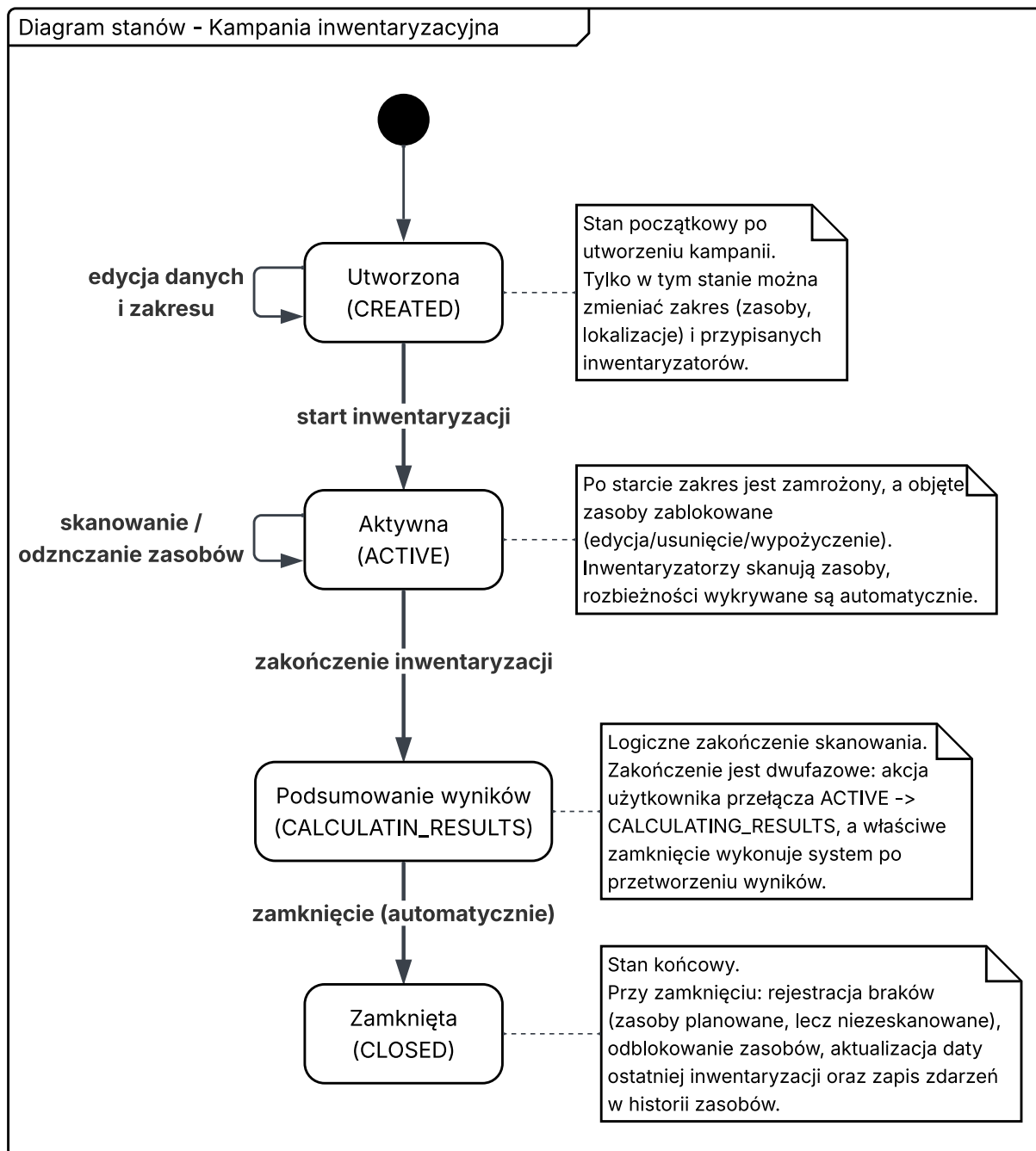
3.9.5 Reporting Service



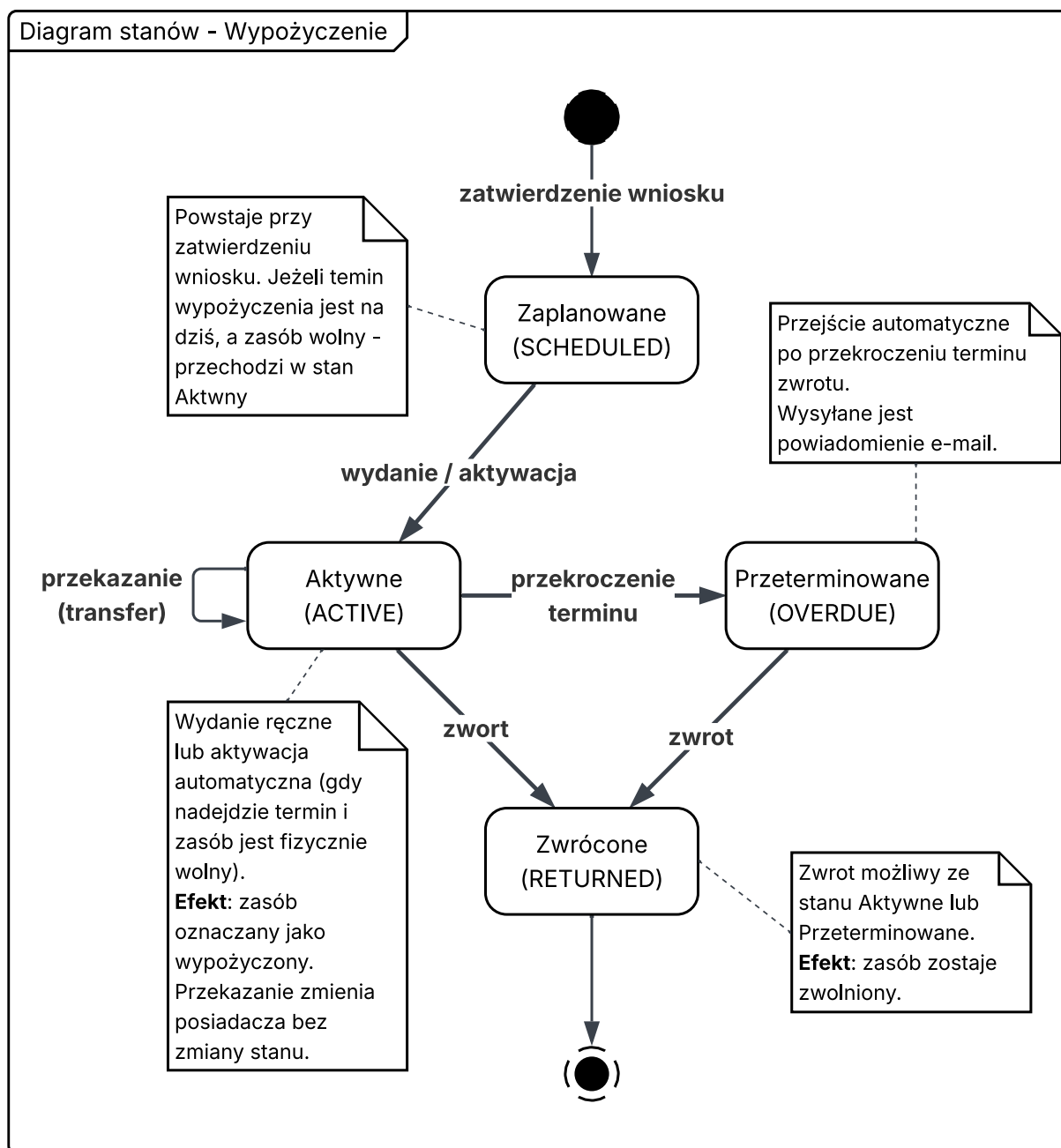
Rysunek 20 Diagram klas mikroserwisu Reporting Service. Źródło: Opracowanie własne.

3.10. Diagramy stanów

W podrozdziale przedstawiono diagramy stanów wybranych bytów systemu, ukazujące ich stany oraz przejścia wraz z warunkami i efektami



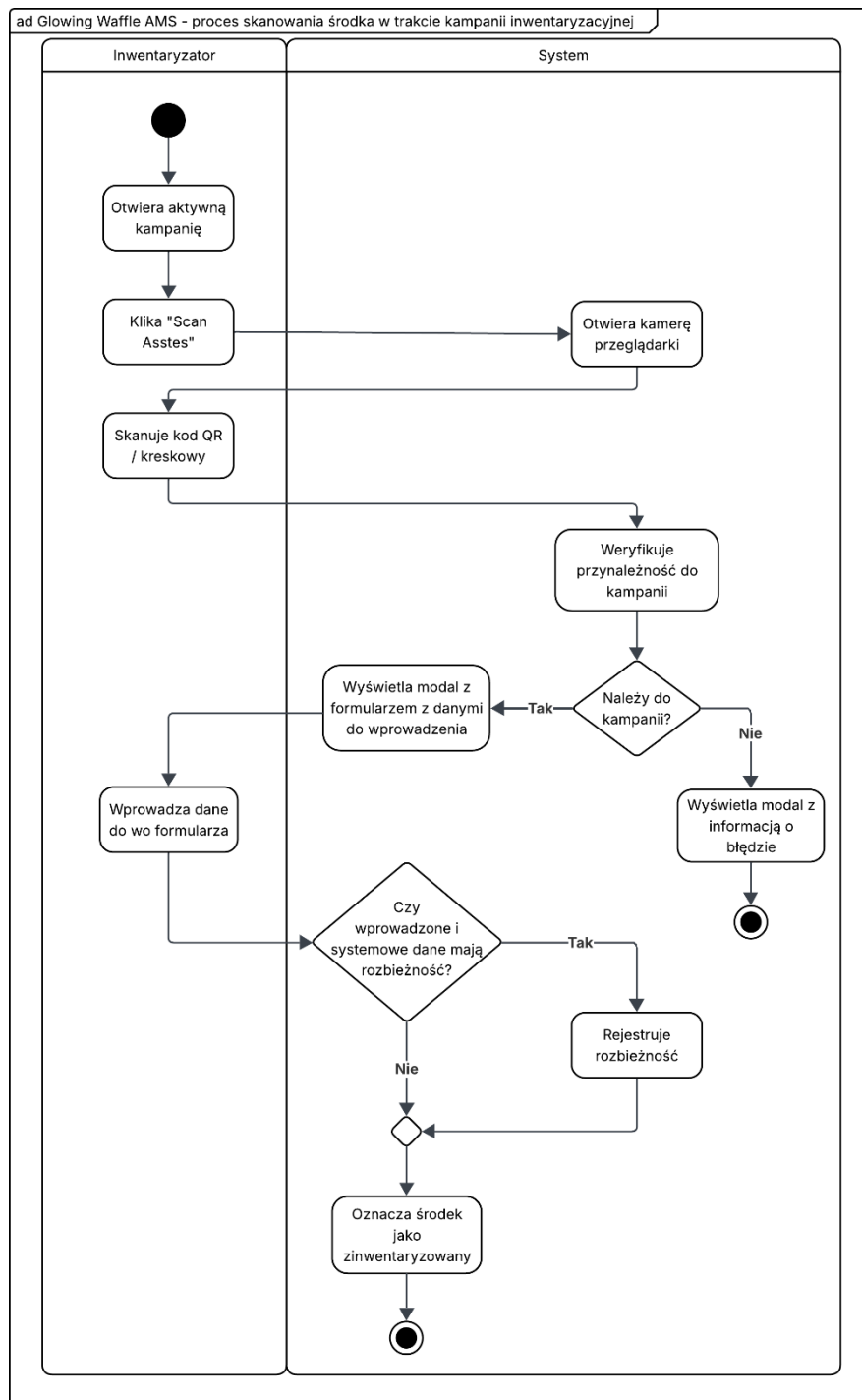
Rysunek 21 Diagram stanów cyklu życia kampanii inwentaryzacyjnej. Źródło: Opracowanie własne



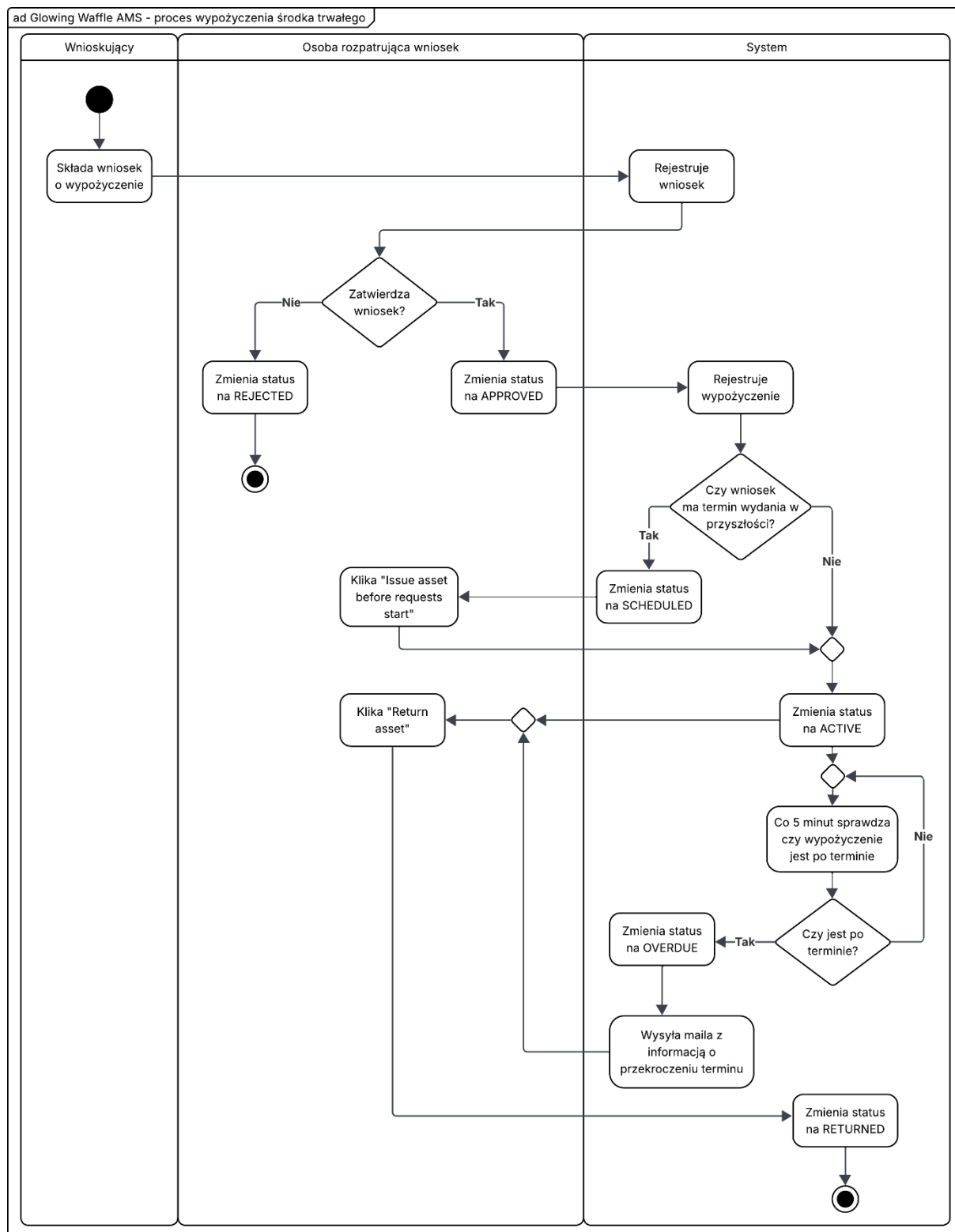
Rysunek 22 Diagram stanów cyklu życia wypożyczenia. Źródło: Opracowanie własne

3.11. Diagramy aktywności

W podrozdziale przedstawiono diagramy aktywności kluczowych procesów systemu, obejmujące przepływ sterowania oraz punkty decyzyjne.



Rysunek 23 Diagram aktywności procesu skanowania środka trwałego w trakcie kampanii inwentaryzacyjnej. Źródło: Opracowanie własne.



Rysunek 24 Diagram aktywności procesu wypożyczenia środka trwałego. Źródło: Opracowanie własne.

3.12.Scenariusze

Podrozdział przedstawia scenariusze najważniejszych przypadków użycia: główny przepływ zdarzeń z perspektywy aktora oraz przebiegi alternatywne.

Tabela 13 Scenariusz przepływu dodania zasobu. Źródło: Opracowanie własne.

Nazwa	Dodanie zasobu	
Aktorzy	Zarządca środków	
Główny przepływ zdarzeń		
L.p.	Nazwa aktywności	Numer następnej aktywności
1	Zarządca środków otwiera moduł zasobów i wybiera „Add Asset"	2
2	Zarządca środków podaje nazwę i wybiera stan zasobu	3
3	Zarządca środków wybiera kategorię oraz opcjonalnie typ	4
4	Zarządca środków wybiera lokalizację	5
5	Zarządca środków wybiera dokument zakupu oraz stawkę VAT	6
6	Zarządca środków podaje cenę jednostkową netto	7
7	Zarządca środków uzupełnia pola niestandardowe kategorii/typu	8
8	Zarządca środków zatwierdza dodanie zasobu	
Alternatywny przebieg zdarzeń		
L.p.	Nazwa aktywności	Numer następnej aktywności
3a	Brakuje wymaganej kategorii, lokalizacji, dokumentu zakupu lub stawki VAT – Zarządca środków tworzy brakujący element	1

Tabela 14 Scenariusz przepływu złożenia wniosku o wypożyczenie. Źródło: Opracowanie własne.

Nazwa	Złożenie wniosku o wypożyczenie	
Aktorzy	Wszyscy pracownicy z dostępem do systemu	
Główny przepływ zdarzeń		
L.p.	Nazwa aktywności	Numer następnej aktywności
1	Pracownik otwiera moduł wypożyczeń i wybiera „Request Equipment"	2
2	Pracownik wybiera zasób lub zasoby do wypożyczenia	3
3	Pracownik określa okres wypożyczenia (datę od i do)	4
4	Pracownik podaje uzasadnienie wniosku	5

5	Pracownik wysyła wniosek	
Alternatywny przebieg zdarzeń		
L.p.	Nazwa aktywności	Numer następnej aktywności
2a	Pracownik wybiera zasób zablokowany trwającą inwentaryzacją – system uniemożliwia jego dodanie	2
3a	Wybrany zasób jest zajęty w żądanym terminie – system oznacza niedostępność. Pracownik zmienia termin lub zasób	3

Tabela 15 Scenariusz przepływu rozpatrzenia wniosku o wypożyczenie. Źródło: Opracowanie własne.

Nazwa	Rozpatrzenie wniosku o wypożyczenie	
Aktorzy	Zarządca wypożyczeń	
Główny przepływ zdarzeń		
L.p.	Nazwa aktywności	Numer następnej aktywności
1	Zarządca wypożyczeń otwiera listę oczekujących wniosków	2
2	Zarządca wypożyczeń wybiera wniosek do rozpatrzenia	3
3	Zarządca wypożyczeń przegląda szczegóły wniosku	4
4	Zarządca wypożyczeń zatwierdza wniosek wraz z komentarzem	
Alternatywny przebieg zdarzeń		
L.p.	Nazwa aktywności	Numer następnej aktywności
4a	Zarządca wypożyczeń odrzuca wniosek wraz z uzasadnieniem	

4. Technologie oraz narzędzia

Rozdział ten szczegółowo opisuje i wprowadza czytelnika w technologie oraz narzędzia informatyczne, których użyto w fazie implementacji projektu Glowing Waffle Asset Management System.

4.1. Java 25

Java jest językiem programowania stworzonym przez amerykańską firmę Sun Microsystems. Swoje początki ma w roku 1991, kiedy James Gosling i Patrick Naughton, postanowili zaproponować język programowania, który mógłby być wydajny i wykorzystywany na dowolnych urządzeniach, niezależnie od ich architektury. Doprowadziło to do utworzenia kompaktowego i przenośnego języka, generującego kod pośredni wykorzystywany przez maszynę wirtualną JVM.

Po kilku latach, dzieło firmy Sun Microsystems ujrzało światło dzienne w magazynie SunWorld w połowie 1995 roku, a kilka miesięcy później, na początku 1996 roku, doczekało się swojej pierwszej publikacji.

Od tego momentu Java była dalej rozwijana przez swoich twórców, aż do 2009 roku, kiedy Sun Microsystems zostało wykupione przez firmę Oracle Corporation, która kontynuowała dalszy jej rozwój. W 2019 roku przyjęty został nowy cykliczny system publikowania kolejnych wersji języka, odbywający się w co półrocznym interwale czasowym [9].

Java posiada wiele cech, które determinują jej wybór jako główny język w systemach logiki aplikacji. Przykłady to:

- Zarządzanie pamięcią – Java pozwala na automatyczne zarządzanie pamięcią operacyjną. Pozwala to na uniknięcie ryzyka związanego z manualnym przydzielaniem oraz zwalnianiem tego zasobu. Zrealizowane to zostało poprzez wprowadzenie mechanizmu o nazwie *garbage collector*, który usuwa z pamięci niepotrzebne obiekty.
- Bogata kolekcja bibliotek oraz frameworków – przez swoją popularność Java doczekała się wielu publicznie [10] dostępnych zasobów, które mają zastosowanie w projektach komercyjnych.
- Niezależność od architektury – główny cel projektowy stojący za stworzeniem Javy, pozwalający na uniwersalne uruchomienie aplikacji na różnych urządzeniach, niezależnie od ich architektury systemowej. Odbywa się to poprzez kompilację programu do pośredniej postaci obsługiwanej przez maszynę wirtualną JVM.
- Obiektowość – jedna z głównych cech Javy, wiążąca się bezpośrednio ze sposobem tworzenia aplikacji. Pozwala ona na organizację kodu w klasy i obiekty, które mogą użytkować mechaniki takie jak dziedziczenie, pozwalające na wysoką modularność i ponowne wykorzystanie elementów aplikacji.

Zważając na wyżej wymienione cechy oraz doświadczenie członków zespołu, zdecydowano o wyborze Javy jako języka programowania niniejszej pracy inżynierskiej. W projekcie wykorzystywana jest Java w wersji 25 z oznaczeniem LTS (*Long-Term Support*), świadcząca o stabilności oraz długoterminowym wsparciu. Cechują ją ulepszenia w kategoriach takich jak wydajność, bezpieczeństwo i optymalizacja.

4.2. Spring

Spring jest frameworkiem dla języka Java, stworzonym z myślą o uproszczeniu procesu budowania złożonych projektów klasy oprogramowania dla przedsiębiorstw. Oferuje kontener aplikacji, którego najpełniejszą implementacją jest interfejs *Application Context*. Odpowiada on za tworzenie komponentów, zarządzania ich cyklem życia oraz powiązanie ich ze sobą w spójną całość. Poszczególne komponenty, określane w nomenklaturze Springa mianem beanów, są łączone przy użyciu wzorca wstrzykiwania zależności (ang. *Dependency Injection*, DI) [10] [11] (str. 487).

Wstrzykiwanie zależności polega na tym, że zamiast samodzielnego tworzenia zależności obiektów przez dany komponent, odpowiedzialność za ich dostarczenie przejmuje kontener. Przekazanie zależności odbywa się najczęściej za pośrednictwem argumentów konstruktora lub metod akcesora. Takie podejście zmniejsza powiązanie między komponentami i zwiększa testowalność kodu, gdyż zależności można swobodnie podmieniać na ich atropy (ang. *mocks*) w testach jednostkowych.

W swoim pierwotnym kształcie *framework* wymagał jawnej, rozbudowanej konfiguracji. Początkowo poprzez pliki XML, a w późniejszych wersjach w postaci klas opatrzonych adnotacją *@Configuration*. Przełomowym okazało się wprowadzenie *frameworka* Spring Boot, stanowiącego rozszerzenie Springa o mechanizm automatycznej konfiguracji (ang. *autoconfiguration*). Na podstawie zawartości ścieżki klas (ang. *classpath*), zmiennych środowiskowych oraz innych czynników, narzędzie to samodzielnie ustala, które biblioteki wymagają konfiguracji, a następnie stosuje ją bez udziału programisty, o ile ten nie zdefiniuje swoich nadpisań [10]. W praktyce oznacza to, że aplikacja gotowa do uruchomienia może wymagać minimalnej ilości kodu konfiguracyjnego, co istotnie przyspiesza proces wytwarzania.

W kontekście systemu Glowling Waffle AMS wybór ekosystemu Spring jako fundamentu warstwy backendowej podyktowany był szeregiem korzyści, które wprost odpowiadały na założenia architektoniczne projektu. Do głównych atutów determinujących wybór tego rozwiązania należą:

- Gotowość do pracy w architekturze mikroserwisowej – dzięki wbudowanym serwerom aplikacyjnym (np. Apache Tomcat), dostarczonym przez Spring Boot, możliwe jest generowanie samowystarczalnych aplikacji. Umożliwiło to uruchomienie każdej z wyodrębnionych usług (np. Asset Service, Loan Service) w postaci niezależnego procesu, co ułatwiło ich późniejszą izolację wewnątrz kontenerów Docker.
- Bogaty ekosystem i integracja poprzez autokonfigurację – wykorzystanie mechanizmów automatycznej konfiguracji pozwoliło na znaczną redukcję tzw. kodu powtarzalnego (ang. *boilerplate code*). W systemie zaimplementowano gotowe, zintegrowane moduły frameworka, takie jak Spring Data JPA do komunikacji z relacyjną bazą danych PostgreSQL, Spring AMQP do obsługi asynchronicznej komunikacji z brokerem RabbitMQ oraz Spring Security współpracujący z serwerem autoryzacji Keycloak.
- Wsparcie dla obserwowalności (ang. *observability*) – ekosystem oferuje moduł Spring Boot Actuator, który w połączeniu z biblioteką Micrometer dostarcza wbudowane mechanizmy monitorowania kondycji aplikacji (ang. *health check*) oraz eksponowania metryk. Pozwoliło to na bezproblemową integrację z systemem Prometheus i realizację śledzenia rozproszonego w standardzie OpenTelemetry.

Zastosowanie technologii z rodziny Spring znacząco zoptymalizowało proces wytwórczy oprogramowania. Zredukowanie złożoności konfiguracyjnej umożliwiło zespołowi skupienie się na implementacji docelowej logiki biznesowej, przy zachowaniu wysokiej skalowalności i niezawodności wymaganej w tworzonym systemie.

4.3. Hibernate

Hibernate jest biblioteką języka Java realizującą mapowanie obiektowo-relacyjne (ang. *Object-Relational Mapping*, ORM). Jej głównym zadaniem jest eliminowanie tzw. niedopasowania impedancji (ang. *impedance mismatch*), czyli strukturalnej różnicy między obiekowym modelem danych języka Java a relacyjnym modelem stosowanym w bazach danych. Programista operuje na obiektach języka Java, zwanych encjami, zaś biblioteka samodzielnie odpowiada za translację operacji wykonywanych na tych obiektach do odpowiednich zapytań SQL [12].

Hibernate stanowi najpopularniejszą implementację specyfikacji JPA (*Java Persistence API*), definiującej standardowy interfejs dostępu do warstwy trwałości danych w ekosystemie Javy. Dzięki temu kod aplikacji pozostaje niezależny od konkretnego dostawcy ORM, co oznacza, że wymiana Hibernate na inną implementację JPA nie wymaga modyfikacji logiki biznesowej.

W systemie Glowing Waffle AMS biblioteka pełni rolę dostawcy JPA i jest wykorzystywana za pośrednictwem modułu Spring Data JPA. Takie ułożenie warstw pozwala na korzystanie z repozytoriów, opartych na interfejsach, gdzie standardowe operacje bazodanowe, jak zapis, odczyt, aktualizacja czy usunięcie encji, dostępne są bez konieczności pisania jawnych zapytań. W przypadkach wymagających bardziej złożonych operacji Hibernate udostępnia język zapytań JPQL (ang. *Java Persistence Query Language*), operujący na encjach i ich polach, a nie bezpośrednio na tabelach i kolumnach bazy danych.

Istotną cechą biblioteki jest mechanizm zarządzania schematem bazy danych, kontrolowany przez właściwość `spring.jpa.hibernate.ddl-auto`. W projekcie zastosowano wartość `update`, przy której Hibernate automatycznie tworzy i aktualizuje strukturę tabel na podstawie zdefiniowanych encji, bez konieczności ręcznego pisania skryptów migracyjnych.

4.4. Maven

Maven jest narzędziem służącym do budowania i zarządzania projektami opartymi na języku Java. Narzędzie powstało jako próba ujednolicenia procesu budowania projektów w ramach inicjatywy Jakarta Turbine, gdzie każdy z podprojektów posiadał własną, nieco odmienną konfigurację procesów kompilacji. Efektem tych prac jest narzędzie umożliwiające standaryzację budowania, zarządzania zależnościami oraz publikowanie informacji o projekcie [13].

Centralnym elementem każdego projektu opartym na Mavenie jest plik `pom.xml` (ang. *Project Object Model*), w którym deklarowane są metadane aplikacji, zależności oraz konfiguracja wtyczek uczestniczących w procesie budowania. Dzięki temu podejściu każdy programista zapoznany z jednym projektem korzystającym z Mavena jest w stanie sprawnie odnaleźć się w dowolnym innym opartym o to narzędzie. W systemie Glowing Waffle AMS plik `pom.xml` definiuje zależności każdego mikroservisu wobec bibliotek ekosystemu Spring, sterownika bazy danych PostgreSQL oraz pozostałych komponentów wykorzystanych w projekcie.

Istotną funkcją tego narzędzia jest automatyczne zarządzanie zależnościami, obejmujące ich pobranie z centralnego repozytorium oraz obsługę zależności przechodnich (ang. *transitive dependencies*), czyli bibliotek wymaganych przez bezpośrednie zależności projektu. Eliminuje to konieczność ręcznego śledzenia i dołączania wymaganych bibliotek.

Maven definiuje również standardowy cykl życia budowania (ang. *build lifecycle*), składający się z uporządkowanych faz, zaczynając od kompilacji kodu źródłowego, przez uruchomienie testów jednostkowych, aż po pakowanie i wdrożenie artefaktów. Wykonanie dowolnej fazy powoduje automatyczne uruchomienie poprzedzających, co zapewnia przewidywalność i powtarzalność całego procesu. W systemie Glowing Waffle AMS każdy mikroservis jest pakowany przez Maven do postaci wykonywalnego pliku JAR, który następnie trafia do obrazu Docker.

Elastyczność narzędzia zapewnia system wtyczek (ang. *plugins*), pozwalający na rozszerzanie i dostosowywanie poszczególnych faz cyklu życia do potrzeb konkretnego projektu. W systemie Glowing Waffle AMS wykorzystano między innymi wtyczkę `spring-boot-maven-plugin`, odpowiedzialną za pakowanie aplikacji do wykonywalnego pliku JAR zawierającego wszystkie zależności, oraz `maven-compiler-plugin`, konfigurującą proces kompilacji z uwzględnieniem procesów adnotacji takich jak Lombok czy MapStruct.

4.5. Baza danych PostgreSQL

PostgreSQL jest obiektowo-relacyjnym systemem do zarządzania bazą danych (ang. *Object-Relational Database Management System*, ORDBMS) o otwartym kodzie źródłowym. Początkami sięga 1986 roku, kiedy powstał jako projekt na Uniwersytecie Kalifornijskim w Berkeley. Przez kilka dekad aktywnego rozwoju system zyskał reputację rozwiązania łączącego niezawodność z szeroką zgodnością ze standardami języka SQL [14].

Jedną z kluczowych cech systemu jest pełna zgodność z własnościami ACID (*Atomicity, Consistency, Isolation, Durability*), gwarantującymi spójność i niezawodność operacji bazodanowych nawet w przypadku awarii. PostgreSQL oferuje również zaawansowane mechanizmy współbieżności oparte na modelu MVCC (*Multi-Version Concurrency Control*), które umożliwia wykonywanie równoległych operacji odczytu i zapisu bez konieczności blokowania zasobów. Przekłada się to bezpośrednio na wydajność systemu przy jednoczesnej obsłudze wielu użytkowników.

Wyróżniającą zaletą PostgreSQL jest rozbudowany system typów danych, który oprócz typów prymitywnych obsługuje struktury złożone, dane geometryczne, a także dokumenty w formacie JSON i JSONB, co czyni go rozwiązaniem uniwersalnym dla różnorodnych zastosowań. Dodatkowo rozszerzalność ta, wiąże się z możliwością definiowania własnych typów danych, funkcji oraz procedur składowych [14].

W zaproponowanym systemie PostgreSQL pełni rolę silnika relacyjnej bazy danych dla każdego z mikroservisów. Zgodnie z przyjętym wzorcem architektonicznym „*Database per service*”, każdy mikroservis posiada własną, izolowaną instancję bazy danych, co wpisuje się w zasadę luźnego powiązania między serwisami omówioną w rozdziale 5.

4.6. TypeScript

TypeScript powstał jako odpowiedź firmy Microsoft na rosnącą trudność utrzymania dużych projektów pisanych w JavaScript [15]. Język ten rozszerza go o statyczny system typów, pozostając jednocześnie w pełni kompatybilny z jego składnią, oznacza to, że każdy poprawnie napisany kod napisany w tym języku jest równocześnie poprawnym kodem TypeScript. Dzieje się tak, ponieważ wynikowy kod kompiluje się z powrotem do czystego JavaScript, co pozwala na jego uruchomienie w dowolnym środowisku obsługującym ten język – zarówno w przeglądarce, jak i po stronie serwera [16].

Kluczową motywacją stojącą za powstaniem języka był problem skalowalności aplikacji pisanych w JavaScript. Wraz ze wzrostem rozmiaru i złożoności projektu brak statycznego typowania prowadził do trudności w wykrywaniu błędów, ujawniających się dopiero podczas działania programu. TypeScript rozwiązuje ten problem poprzez wprowadzenie statycznej analizy kodu, która wykrywa nieprawidłowe użycie typów jeszcze przed uruchomieniem aplikacji, a dokładnie na etapie kompilacji [17]. Przekłada się to na wyższą jakość kodu oraz skrócenie czasu potrzebnego do diagnozy usterek.

System typów języka jest opcjonalny i elastyczny, gdzie możliwe jest stopniowe wprowadzanie adnotacji typów do istniejącej bazy kodu JavaScript, bez konieczności przepisywania całego projektu. Język oferuje zarówno typy prymitywne, jak i mechanizmy definiowania złożonych struktur danych

przy pomocy interfejsów (ang. *interface*) oraz aliasów typów (ang. *type aliases*) [18]. Uzupełnieniem systemu typów jest mechanizm wnioskowania o typach (ang. *type inference*) [19], który w wielu przypadkach pozwala kompilatorowi samodzielnie określić typ zmiennej bez jawnej adnotacji ze strony programisty.

System Glowing Waffle AMS wykorzystuje TypeScript wyłącznie do warstwy interfejsu użytkownika projektu. Jego zastosowanie w połączeniu z frameworkiem Next.js, który szerzej opisywany jest w rozdziale 4.7. pozwoliło na precyzyjne zdefiniowanie kontraktów między komponentami interfejsu a warstwą komunikacyjną z serwerem, co zwiększyło czytelność i bezpieczeństwo kodu.

4.7. Next.js

Next.js jest frameworkiem oparty na bibliotece React, przeznaczonym do budowania pełnych aplikacji webowych. Rozszerza on możliwości Reacta o dodatkowe mechanizmy optymalizacji oraz automatycznie konfiguruje narzędzia niższego poziomu, takie jak bundlery i kompilatory, jednocześnie zwalniając programistę z obowiązku ręcznego doboru i konfiguracji [20].

Jedną z głównych cech frameworka jest elastyczny model renderowania. Next.js umożliwia stosowanie renderowania po stronie serwera (ang. *Server-Side Rendering*, SSR), statycznego generowania stron (ang. *Static Site Generation*, SSG) oraz renderowania po stronie klienta (ang. *Client-Side Rendering*, CSR), przy czym wybór strategii może być podejmowany niezależnie dla każdej strony aplikacji. Pozwala to na dostosowanie podejścia do charakteru konkretnego widoku, gdzie strony wymagające aktualnych danych mogą korzystać z SSR, natomiast strony o stabilnej treści mogą być generowane statycznie, co przekłada się na lepszą wydajność i krótszy czas ładowania.

Ważnym elementem architektury Next.js jest system wyznaczania trasy (ang. *routing*) oparty na strukturze systemu plików. Każdy plik umieszczony w odpowiednim katalogu projektu automatycznie staje się dostępną ścieżką aplikacji, bez konieczności jawnej konfiguracji tras. Framework oferuje dwa warianty wyznaczania tras – *App Router*, wpierający nowe mechanizmy Reacta, w tym komponenty serwerowe (ang. *Server Components*), oraz starszy *Page Router*, który dalej jest utrzymywany i rozwijany.

W projekcie systemu Next.js pełni główną rolę frameworka warstwy interfejsu użytkownika, z wykorzystaniem nowszego *App Router*. Zastosowanie komponentów serwerowych pozwoliło na przeniesienie części logiki pobierania danych na stronę serwera, co zmniejszyło ilość kodu wykonywanego po stronie klienta i przyspieszyło czas pierwotnego renderowania interfejsu użytkownika. Mechanizm *Route Handlers*, dostępny w katalogu `app/api`, posłużył do zdefiniowania warstwy pośredniej między częścią interfejsu a mikroserwisami, gdzie dla każdego z nich wyodrębniono osobny zestaw końcówek (ang. *endpoints*), co szczegółowo omówiono w rozdziale 6.

W konfiguracji projektu zastosowano tryb output: „standalone”, który podczas budowania generuje zoptymalizowany pakiet zawierający wyłącznie pliki niezbędne do uruchomienia aplikacji. Rozwiązanie to bezpośrednio wspiera konteneryzację, gdzie bardziej szczegółowa konfiguracja środowiska Docker opisana jest w podrozdziale 4.9. Warstwa interfejsu uczestniczy również w ogólnej strategii obserwowalności systemu, do której użyto bibliotekę `@vercel/otel`, gdzie skonfigurowano śledzenie rozproszone po stronie klienta, co szerzej omówiono w podrozdziale 4.17.

4.8. RabbitMQ

RabbitMQ jest otwartoźródłowym brokerem (pośrednikiem) wiadomości, napisanym w języku Erlang, którego model współbieżności dopasowany jest do obsługi wielu równoległych połączeń

sieciowych. Narzędzie zostało opublikowane w roku 2007 przez firmę Rabbit Technologies, a obecnie rozwijane w ramach grupy Broadcom [21].

Podstawowym protokołem komunikacyjnym brokera jest AMQP (*Advanced Message Queuing Protocol*). Jego centralnym elementem jest rozróżnienie między wymiennikiem (ang. *exchange*) a kolejką (ang. *queue*). Producent wiadomości (ang. *producer*) nigdy nie kieruje jej bezpośrednio do kolejki – natomiast publikuje ją do wymiennika, który na podstawie zdefiniowanych reguł routingu i powiązań (ang. *bindings*) przekierowuje wiadomość do właściwych kolejek. Konsument (ang. *consumer*) subskrybuje daną kolejkę i odbiera wiadomości w miarę ich napływu. Wyróżnia się dwa modele dostarczania wiadomości do konsumenta:

- model *push* (subskrypcja) – broker samodzielnie przekazuje wiadomości nasłuchującemu konsumentowi zaraz po ich pojawieniu się w kolejce,
- model *pull* (odpytywanie) – to konsument aktywnie żąda pojedynczej wiadomości w wybranym przez siebie momencie.

RabbitMQ definiuje również cztery typy wymienników:

- *direct* (komunikacja bezpośrednia) – wiadomości trafiają tylko do kolejki, w której klucz routingu (ang. *routing key*) zgadza się z kluczem wystawionym przez producenta wiadomości,
- *fanout* (subskrypcja – *broadcast*) – wiadomość jest rozsyłana do wszystkich powiązanych z danym wymiennikiem kolejek, bez zdefiniowania klucza routingu,
- *topic* (rozgłaszanie selektywne – *multicast*) – komunikaty są kierowane do kolejek na podstawie dopasowania wzorca klucza routingu, gdzie symbol „***” zastępuje dokładnie jedno słowo, a symbol „*#*” dowolną ich liczbę,
- *headers* (nagłówki) – wiadomości kierowane są na podstawie nagłówków wiadomości, a nie samych kluczy routingu.

Broker oferuje szereg mechanizmów zapewniających niezawodność dostarczania wiadomości. Potwierdzenia konsumenta (ang. *consumer acknowledgements*) pozwalają brokerowi na ponowne dostarczenie wiadomości w przypadku awarii. Trwałość (ang. *durability*) kolejek i wiadomości umożliwia restart brokera bez utraty danych. Uzupełnieniem jest mechanizm martwych wiadomości (ang. *Dead Letter Queue*, DLQ) – wiadomości niedostarczone lub odrzucone przez konsumenta mogą być automatycznie przekierowane do dedykowanej kolejki przeznaczonej do analizy i ponownego przetworzenia. RabbitMQ dostarcza również wbudowaną wtyczkę zarządzającą (ang. *management plugin*), umożliwiającą monitorowanie stanu kolejek, wymienników, powiązań i połączeń w czasie rzeczywistym za pomocą interfejsu w przeglądarce.

W systemie Glowing Waffle AMS RabbitMQ zapewnia wewnętrzną komunikację asynchroniczną między mikroservisami. Integracja z warstwą backendową opartą na frameworku Spring Boot odbywa się za pośrednictwem biblioteki Spring AMQP, która dostarcza model konfiguracji kolejek i wymienników, oraz mechanizm adnotacji `@RabbitListener` umożliwiający rejestrowanie konsumentów [22][23]. Szczegółowe uzasadnienie wyboru RabbitMQ omówione zostało w podrozdziale 5.2, natomiast szczegóły implementacji opisano w rozdziale 6.

4.9. Docker

Docker jest platformą, która umożliwia pakowanie aplikacji wraz z całym ich środowiskiem uruchomieniowym, w których znaleźć mogą się biblioteki, zależności i konfiguracje do przenośnych i izolowanych jednostek. Takie podejście eliminuje problem rozbieżności między środowiskiem deweloperskim a produkcyjnym, skracając tym samym czas potrzebny na wdrożenie oprogramowania [24].

Centralnym pojęciem platformy jest kontener (ang. *container*), czyli działająca instancja obrazu (ang. *image*), zawierająca wszystkie elementy potrzebne do uruchomienia aplikacji. Obraz jest szablonem tylko do odczytu, definiującym zawartość kontenera i budowanym na podstawie pliku Dockerfile. Każda instrukcja w tym pliku tworzy osobną warstwę obrazu, co sprawia, że ponowne budowanie wymaga przetworzenia wyłącznie zmienionych warstw. Czyni to obrazy lekkimi i szybkimi w budowaniu w porównaniu do tradycyjnych maszyn wirtualnych [24].

Docker działa w architekturze klient-serwer. Demon Dockera (ang. *Docker daemon*) odpowiada za budowanie, uruchamianie i dystrybucję kontenerów, natomiast klient (Docker) jest interfejsem wiersza poleceń służącym do wydawania poleceń demonowi poprzez REST API. Uzupełnieniem ekosystemu jest Docker Compose, które jest narzędziem pozwalającym na deklaratywne definiowanie i uruchamianie środowisk złożonych z wielu kontenerów przy pomocy pojedynczego pliku konfiguracyjnego w formie YAML.

W systemie Glowing Waffle AMS Docker stanowi podstawę środowiska uruchomieniowego. Każdy mikroserwis posiada własny plik Dockerfile, w którym zastosowano wieloetapowe budowanie obrazu (ang. *multi-stage build*). W etapie pierwszym obraz oparty na Maven kompiluje kod źródłowy i buduje plik wykonywalny JAR. Etap drugi korzysta z minimalnego obrazu bazowego distroless, zawierającego wyłącznie środowisko uruchomieniowe Java – bez powłoki systemowej ani zbędnych narzędzi, co zmniejsza powierzchnię ataku oraz rozmiar końcowego obrazu. Zarządzanie całym środowiskiem, czyli mikroserwisami, bazami danych, brokerem RabbitMQ, serwerem Keycloak oraz warstwą monitoringu, odbywa się przy pomocy Docker Compose, co szczegółowo omówiono w podrozdziale 6.5.

4.10. Traefik

Traefik jest otwartoźródłowym serwerem *reverse proxy* oraz bramą API, napisanym w języku Go. Został opublikowany w 2015 roku przez Emile’a Vauge’a, a obecnie projekt rozwijany jest przez firmę Traefik Labs (wcześniej pod nazwą Containous) założoną przez twórcę Traefika. Narzędzie powstało głównie z myślą o środowiskach kontenerowych (Kubernetes, Docker), w odpowiedzi na problem dynamicznego routingu w takich środowiskach. W odróżnieniu od klasycznych serwerów typu *reverse proxy*, takich jak Nginx czy HAProxy, w których reguły routingu wymagają ręcznego utrzymywania osobnego pliku konfiguracyjnego, Traefik automatycznie wykrywa uruchomiane usługi i konfiguruje na ich podstawie routery, eliminując konieczność przeładowywania konfiguracji przy każdej zmianie stanu środowiska [25].

Kluczowym mechanizmem konfiguracyjnym narzędzia jest system tak zwanych dostawców (ang. *providers*), pełniących rolę adapterów do różnych źródeł danych o uruchomionych usługach – między innymi platformy Docker, klastra Kubernetes, czy zwykłych plików konfiguracyjnych. W modelu opartym na Dockerze, konfiguracja routerów oraz usług definiowana jest za deklaratywne pomocą etykiet (ang. *labels*) przypisanych do poszczególnych kontenerów. Traefik nasłuchuje zdarzeń demona Dockera (ang. *Docker daemon*) i w czasie rzeczywistym aktualizuje stan wewnętrznych reguł routingu. Wśród pozostałych wbudowanych funkcji narzędzia można wymienić obsługę protokołu ACME (*Automatic Certificate Management Environment*), pozwalającą na automatyczne wystawianie i odnawianie certyfikatów TLS (ang. *Transport Layer Security* – protokół kryptograficzny zapewniający szyfrowane i uwierzytelnione połączenie między klientem a serwerem) za pośrednictwem usług takich usług jak Let’s Encrypt. Należy również wymienić mechanizm warstw pośredniczących (ang. *middleware*), umożliwiający nakładanie dodatkowych operacji na trasę żądania, takich jak ograniczenie tempa zapytań, przepisywanie nagłówków czy wymuszenie uwierzytelnienia. Do istotnych funkcji należą: panel administracyjny prezentujący stan zdefiniowanych routerów oraz usług, a także natywne eksporty m.in. metryk i logów w standardzie OpenTelemetry oraz Prometheus.

W systemie Glowing Waffle AMS wykorzystano Traefika jako bramę API obsługującą cały ruch przychodzący do platformy. Konfiguracja routerów oraz usług definiowana jest poprzez etykiety kontenerów mikroservisów, ich publiczne nazwy domenowe (np. `assets-app.glowingwaffle.app`) oraz parametry obsługi TLS. Certyfikaty TLS dla wszystkich publicznych domen aplikacji wystawiane są automatycznie przez wbudowany *resolver* ACME. Eksport metryk, śladów rozproszonych oraz logów Traefika kierowany jest do kolektora OpenTelemetry, którego rolę w warstwie obserwowalności systemu opisano w podrozdziale 5.7.

4.11.Keycloak

Keycloak jest otwartoźródłowym serwerem do zarządzania tożsamością i dostępem (ang. *Identity and Access Management*), napisanym w języku Java. Projekt został zapoczątkowany w 2014 roku przez firmę Red Hat, a w 2023 roku system został przekazany pod skrzydła fundacji *Cloud Native Computing Foundation* (CNCF), pod którymi do dziś jest rozwijany. Nadrzędnie realizuje funkcje uwierzytelniania (ang. *authentication*) oraz dostarcza dane autoryzacyjne (role) w sposób scentralizowany, zdejmując z poszczególnych aplikacji obowiązek samodzielnego zarządzania poświadczeniami użytkowników. Keycloak udostępnia natywną implementację branżowych standardów bezpieczeństwa, w tym OAuth 2.0, OpenID Connect (OIDC) oraz SAML 2.0 [26].

Podstawową jednostką organizacyjną serwera jest dziedzina (ang. *realm*) – odizolowana przestrzeń grupująca użytkowników, role, grupy oraz aplikacje klienckie które nie współdzielą informacji z innymi dziedzinami. W obrębie danej dziedziny definiuje się klientów (ang. *clients*), reprezentujących aplikacje delegujące proces uwierzytelniania do Keycloak, a także wymienione już wcześniej role i grupy, na których serwer opiera model kontroli dostępu. Istotną cechą narzędzia jest mechanizm federacji użytkowników (ang. *user federation*), umożliwiający integrację z zewnętrznymi magazynami tożsamości – takimi jakimi są LDAP oraz Microsoft Active Directory, bez konieczności ręcznego zakładania i utrzymywania kont w systemie – przy zachowaniu katalogu LDAP jako nadrzędnego źródła prawdy. Uzupełnieniem jest wsparcie dla pośredniczenia tożsamości (ang. *identity brokering*), umożliwiające delegowanie uwierzytelnienia do zewnętrznych dostawców tożsamości (np. Google, GitHub).

Z perspektywy aplikacji klienckich Keycloak pełni rolę serwera autoryzacji oraz dostawcy tożsamości (ang. *OpenID Provider*). Po pomyślnym uwierzytelnieniu wystawia on tokeny w formacie JWT, w tym token dostępu (ang. *access token*) który aplikacja dołącza do każdego wysyłanego żądania w celu potwierdzenia tożsamości i uprawnień użytkownika. Poza nim wystawia również token odświeżający oraz identyfikacyjny, służące do odnowienia tokenu dostępu oraz identyfikacji konta. Klucze publiczne służące do weryfikacji podpisu tokenów udostępniane są przez publiczną końcówkę JWKS (*JSON Web Key Set*), co pozwala aplikacji weryfikować tokeny lokalnie, bez odpytywania serwera przy każdym żądaniu. Administrowaniem dziedzin, klientów i użytkowników można zarządzać zarówno z poziomu graficznej konsoli administracyjnej, jak i programowo, poprzez wystawienie przez serwer Admin API.

W systemie Glowing Waffle AMS Keycloak pełni rolę centralnego serwera autoryzacji i uwierzytelnienia. Na potrzeby systemu utworzono dedykowaną dziedzinę `glowing-waffle-ams` wraz z klientem `nextjs-app` obsługującym aplikację frontendową. Konta użytkowników zaczytywane są z zewnętrznego katalogu LDAP za pośrednictwem skonfigurowanej federacji użytkowników, co odpowiada założeniu integracji z istniejącą infrastrukturą organizacji. Wyodrębniony mikroservis `kc-proxy` komunikuje się z Keycloak Admin API w celu pozyskania oraz udostępnianie reszcie mikroservisów informacji o kontach użytkowników. Szczegółowe omówienie przepływu autoryzacyjnego *Authorization Code Flow* z rozszerzeniem PKCE, czy struktury tokenów JWT przedstawiono w podrozdziale 5.4 niżej.

4.12. Git

Git jest rozproszonym systemem kontroli wersji (ang. *Distributed Version Control System*, DVCS), umożliwiającym śledzenie historii zmian w plikach oraz koordynowaniu prac wielu osób nad wspólną bazą kodu. Początkowo system ten powstał z potrzeby zarządzania kodem źródłowym jądra Linux, który zrzeszał programistów z całego świata. Wymagania stawiane przez ten projekt, ukształtowały Git jako narzędzie, w którym największymi priorytetami są wydajność i niezawodność [27]. Według danych z ankiety Stack Overflow przeprowadzonej w 2022 roku, Git używany jest przez ponad 95% zawodowych programistów, co czyni go standardem w branży [27].

Kluczową cechą odróżniającą Git od starszych, scentralizowanych systemów kontroli wersji jest rozproszony model repozytoriów. Każdy użytkownik posiadający dostęp do projektu, ma również pełną kopię historii zmian na własnym urządzeniu, co umożliwia pracę w trybie odłączenia od sieci (ang. *offline*) i eliminuje pojedynczy punkt awarii charakterystyczny dla rozwiązań scentralizowanych. Zmiany są zapisywane w postaci migawek (ang. *snapshots*) całego stanu repozytorium, a nie różnic między plikami, co przekłada się na wydajność większości operacji [28].

W przedstawionym projekcie Git pełni rolę systemu kontroli wersji dla całej bazy kodu projektu. Repozytorium hostowane jest na platformie GitHub, gdzie każdy mikroserwis oraz warstwa frontendowa są rozwijane w ramach wspólnego repozytorium typu monorepo. Gałęzie (ang. *branches*) służą do izolowania prac nad nowymi funkcjonalnościami oraz testowaniem owych zmian. Scalanie kodu odbywa się poprzez mechanizm pull requestów, co pozwala na wzajemną weryfikację kodu przez członków zespołu przed włączeniem zmian do głównej gałęzi projektu.

4.13. Github Actions

GitHub Actions jest platformą wbudowaną bezpośrednio w serwis GitHub, służącą do automatyzacji procesów ciągłej integracji i ciągłego dostarczania oprogramowania (ang. *Continuous Integration / Continuous Delivery*, CI/CD). Pozwala na definiowanie zautomatyzowanych przepływów pracy (ang. *workflows*), reagujących na zdarzenia zachodzące w repozytorium – takie jak scalanie gałęzi czy otwarcie pull requesta [29].

Podstawowym elementem konfiguracyjnym jest plik YAML umieszczony w katalogu `.github/workflows`, opisujący jeden lub więcej zadań (ang. *jobs*). Każde zadanie składa się z kroków (ang. *steps*), wykonywanych sekwencyjnie na wirtualnej maszynie zwanej runnerem. Kroki mogą uruchamiać dowolne skrypty powłoki lub korzystać z gotowych wielokrotnego użytku akcji (ang. *actions*) udostępnionych przez społeczność platformy GitHub Marketplace. Zadania mogą być wykonywane równolegle bądź zależnie od siebie, co pozwala na modelowanie złożonych potoków wytwórczych [29].

System Glowing Waffle AMS wykorzystuje GitHub Actions w trzech obszarach automatyzacji. Pierwszym z nich jest potok CI/CD budujący i publikujący obrazy Docker dla wszystkich mikroserwisów. Przepływ ten jest wyzwalany przy każdym scalaniu zmian z gałęzią główną i wykorzystuje mechanizm macierzy (ang. *matrix*) do równoległego budowania obrazów wszystkich mikroserwisów. Zbudowane obrazy są publikowane do rejestru GitHub Container Registry, skąd pobierane są w kolejnym etapie przez potok wdrożeniowy. Drugi przepływ odpowiada za wdrażanie środowiska produkcyjnego – po pomyślnym zakończeniu potoku budowania, plik `docker-compose.prod.yaml` wraz z plikami konfiguracyjnymi kopiowany jest na serwer produkcyjny przez SSH, gdzie następuje podstawienie zmiennych środowiskowych z zaszyfrowanych sekretów repozytorium oraz uruchomienie zaktualizowanego środowiska. Trzecim obszarem jest Dependabot – mechanizm automatycznego wykrywania i zgłaszania dostępnych aktualizacji zależności, szczególnie przydatny w warstwie frontendowej ze względu na częste aktualizacje bibliotek ekosystemu React i Next.js.

4.14. Postman / Insomnia

Postman jest aplikacją klienta usług sieciowych (ang. *API client*) służącą do wywoływania i testowania interfejsów HTTP. Narzędzie umożliwia konstruowanie żądań HTTP z dowolną metodą (GET, POST, PUT, PATCH, DELETE), wraz z nagłówkami, parametrami zapytania oraz ciałem komunikatu, a następnie inspekcję odpowiedzi serwera – kodu statusu, nagłówków, oraz treści odpowiedzi w różnych formatach m.in. JSON, XML. Utworzone żądania mogą być zapisane w programie i grupowane w tzw. kolekcje (ang. *collections*), stanowiące logiczną strukturę katalogową ułatwiającą ich organizację oraz wielokrotne wykorzystywanie [30].

W początkowej fazie wytwarzania systemu Glowing Waffle AMS Postman wspomagał ręczne testowanie końcówek REST udostępnionych przez poszczególne mikroserwisy. Zespół utrzymywał kolekcję żądań odpowiadającą zaimplementowanym końcówkom backendu, wykorzystywaną zarówno przez programistów warstwy serwerowej do weryfikacji poprawności implementacji, jak i przez osoby pracujące nad warstwą frontendową – celem rozpoznania struktury danych wymaganych oraz zwracanych przez API. Kolekcja ta nie pełniła roli formalnej dokumentacji interfejsu, którą powierzono narzędziu Swagger, opisanemu w podrozdziale 4.16.

Wraz z postępem prac pojawiła się potrzeba współdzielenia kolekcji żądań pomiędzy członkami zespołu w trybie pozwalającym na jej wspólną edycję. Model współdzielenia przestrzeni roboczej (ang. *workspace*) w darmowej edycji Postmana okazał się jednak zbyt restrykcyjny, aby umożliwić tego rodzaju kolaborację bez przejścia na płatny plan [31]. Z tego względu zespół zdecydował się na migrację do Insomni – funkcjonalnie zbliżonego klienta API, rozwijanego przez firmę Kong Inc. i dostępnego w darmowej wersji desktopowej [32].

Insomnia oferuje funkcje analogiczne do Postmana w zakresie konstruowania żądań HTTP oraz organizowania ich w kolekcje, dodatkowo udostępnia mechanizmy ułatwiające pracę z dynamicznymi danymi testowymi. W toku prac nad systemem skorzystano z udostępnionych przez oprogramowanie funkcji, przede wszystkim z generatorów wartości losowych (ang. *fakers*) umożliwiających automatyczne podstawianie danych takich jak daty, identyfikatory UUID, czy liczby. W późniejszej fazie rozwoju systemu Glowing Waffle AMS w dużym stopniu został używany mechanizm referencji odpowiedzi (ang. *response chaining*) pozwalający na pobieranie wartości z odpowiedzi wcześniej wykonanych żądań i wstawianie ich do wartości ciała kolejnych żądań. Mechanizmy te okazały się szczególnie przydatne przy testowaniu scenariuszy obejmujących wiele wywołań – na przykład uruchomienia żądania startu wcześniej utworzonej kampanii inwentaryzacyjnej bez ręcznego pozyskiwania i wklejania identyfikatora UUID tejże kampanii.

Z perspektywy projektu istotne znaczenie miało również natywne wsparcie obu narzędzi dla mechanizmów autoryzacji żądań, a w szczególności dla standardu OAuth 2.0. Konfiguracja autoryzacji dla kolekcji bądź pojedynczego żądania obejmuje wybór przepływu autoryzacyjnego (np. *Authorization Code Flow*, *Client Credentials Flow*), wskazanie końcówek serwera autoryzacji oraz parametrów klienta, na podstawie których narzędzie samodzielnie przeprowadza proces pozyskania tokenu dostępu. Jest on następnie automatycznie dołączany do nagłówka *Authorization* dla żądań dziedziczących konfigurację autoryzacji z poziomu kolekcji, do której zostały przypisane. W systemie Glowing Waffle AMS, w którym warstwa bezpieczeństwa oparta jest na serwerze Keycloak realizującym przepływ *Authorization Code Flow* z rozszerzeniem PKCE (opisanym szczegółowo w podrozdziale 5.4), wsparcie to istotnie usprawniło proces ręcznego testowania końcówek REST. Bez tego mechanizmu każdorazowe wywołanie API wymagałoby ręcznego przeprowadzania procesu logowania, pobrania tokenu uwierzytelniającego oraz jego wstawienia w nagłówek żądania.

Warto również wspomnieć o pozostałych mechanizmach oferowanych przez opisywane narzędzia, znacząco rozszerzających możliwości pracy z kolekcją żądań. Pierwszym z nich jest konfiguracja wielu środowisk uruchomieniowych (ang. *environments*) pozwalająca na parametryzację żądań w zależności od kontekstu uruchomienia – przykładowo poprzez zdefiniowanie odrębnych adresów bazowych oraz danych uwierzytelniających dla środowiska deweloperskiego i produkcyjnego,

przełączanych w obrębie tej samej kolekcji. Drugim z nich są zautomatyzowane skrypty testowe wykonywane w ramach kolekcji, umożliwiające weryfikujące odpowiedzi serwera – dla przykładu: oczekiwany kod statusu, obecność wybranych pól w ciele odpowiedzi czy zgodność wartości z założonym wzorcem.

4.15.Discord

Discord jest platformą komunikacji tekstowej, głosowej i wideo, stworzoną w 2015 roku przez firmę Discord Inc. Pierwotnie skierowana do społeczności graczy, z biegiem czasu zyskała uznanie i popularność w szerszej grupie odbiorców m.in. w kontekstach edukacyjnych oraz zawodowych – w szczególności w mniejszych zespołach projektowych. Podstawową jednostką organizacyjną jest serwer, w obrębie którego uprawnieni użytkownicy mogą tworzyć kanały tematyczne (ang. *channels*) o charakterze tekstowym i głosowym. Kanały można przyporządkowywać do kategorii, co ułatwia zachowanie porządku i tematyczne grupowanie konwersacji. Kanały tekstowe wspomagane przez formatowanie tekstu (ang. *markdown*) wspierają wymianę wiadomości wzbogaconych m.in. o załączniki, bloki kodu z podświetleniem składni, pogrubienia, kursywę czy nagłówki. Kanały głosowe natomiast umożliwiają prowadzenie rozmów wraz z opcjonalnym udostępnianiem ekranu [33].

W ramach projektu Discord pełnił rolę głównej platformy do komunikacji zespołu. W obrębie utworzonego serwera wyodrębniono kanały tekstowe skategoryzowane pod poszczególne obszary prac – w tym osobne kanały dla warstwy backendowej, warstwy frontendowej, zgłoszeń błędów oraz ogólnej komunikacji członków zespołu – co ułatwiało porządkowanie dyskusji oraz odnajdywanie ważnych wątków i wcześniejszych ustaleń. Kanały głosowe wykorzystywane były do bieżących spotkań zespołu, służących omawianiu postępów prac, napotkanych problemów oraz wymianie koncepcji, a także do wspólnych przeglądów kodu (ang. *code review*) z wykorzystaniem mechanizmu współdzielenia ekranu. Do serwera zespołowego dołączono również bota MyRepoBot for Discord, rozwijanego przez firmę Redwerk, integrującego serwer Discord z repozytorium kodu umieszczonego na platformie GitHub. Do zadań MyRepoBot należało publikowanie na wydzielonym kanale tekstowym powiadomień o cyklu życia zgłoszeń scalania (ang. *pull request*) – w tym o ich utworzeniu, zaktualizowaniu, scaleniu oraz zamknięciu [34]. Mechanizm ten redukował konieczność manualnego śledzenia stanu repozytorium oraz przyspieszał czas reakcji członków zespołu na zdarzenia zachodzące w zdalnym repozytorium. O wyborze Discorda spośród alternatywnych komunikatorów zdecydowała prostota obsługi, przejrzystość interfejsu, integracja kluczowych funkcji komunikacyjnych w jednym narzędziu, a także popularność wśród społeczności programistycznej, dzięki której platforma wykształciła ekosystem integracji z innymi systemami oraz dojrzały zbiór botów rozszerzających jej funkcjonalność.

4.16.Swagger

Swagger jest zastawem narzędzi o otwartym dostępie do kodu źródłowego (ang. *open-source*) zbudowanych wokół specyfikacji OpenAPI (ang. *OpenApi Specification*), służących do opisywania interfejsów REST API. Specyfikacja ta pozwala na opis struktury API, niezależnie od wykorzystanego języka programowania, w którym zawarte są informacje takie jak dostępne ścieżki i operacja, parametry żądań formatów odpowiedzi oraz stosowanych mechanizmów uwierzytelnienia. Pliki konfiguracji mogą być tworzone w formacie YAML lub JSON [35].

Z perspektywy projektu kluczowym narzędziem wchodzącym w skład ekosystemu Swagger jest narzędzie Swagger UI, czyli komponent renderujący definicję OpenAPI w postaci interaktywnej dokumentacji dostępnej w przeglądarce. Umożliwia ona przeglądanie dostępnych końcówek oraz wykorzystanie testowych zapytań, bezpośrednio z poziomu interfejsu narzędzia, bez konieczności korzystania z zewnętrznych narzędzi.

W systemie Glowing Waffle AMS dokumentacja API generowana jest automatycznie przy użyciu biblioteki `springdoc-openai`, która na podstawie adnotacji i struktury kontrolerów tworzy definicję OpenAPI w czasie działania aplikacji. Każdy mikroservis udostępnia własną instancję Swagger UI pod dedykowaną ścieżką, co pozwala na niezależne przeglądanie i testowanie końcówek poszczególnych serwisów. Narzędzie to pełniło istotną rolę w procesie wywarzania aplikacji – stanowiło formowalne źródło dokumentacji interfejsów backendowych w kontekście pracy nad warstwą frontendową, uzupełniając tym samym ręczne testowanie realizowane przy pomocy aplikacji Insomnia, opisanej w podrozdziale 4.14.

4.17. OpenTelemetry

OpenTelemetry jest otwartym frameworkiem oraz zestawem narzędzi służącym do zbierania, przetwarzania i eksportowania trzech podstawowych rodzajów danych telemetrycznych: śladów (ang. *traces*), metryk (ang. *metrics*) oraz dzienników zdarzeń (ang. *logs*) [36]. Projekt jest niezależny od dostawcy, co oznacza, że debrane dane mogą być przekazywane do dowolnego systemu backendowego, takie jak Jaeger, Prometheus czy Grafana, bez konieczności zmiany kodu instrumentacyjnego.

Zaletą standardu jest to, że aplikacja wymaga instrukcji tylko raz, a dalszy wybór narzędzi analitycznych pozostaje decyzją operacyjną, nie wymagającą modyfikacji kodu źródłowego. OpenTelemetry definiuje wspólny protokół przesyłania danych – OTLP (*OpenTelemetry Protocol*) – oraz semantyczne konwencje nazewnictwa, które ujednolicają sposób opisywania danych telemetrycznych niezależnie od języka programowania czy środowiska uruchomieniowego [36].

W przedstawionym systemie instrumentacja mikroservisów backendowych jest realizowana przy pomocy biblioteki `micrometer-registry-otlp`, eksportującej metryki w protokole OTLP. Dane telemetryczne ze wszystkich serwisów trafiają do centralnego komponentu pośredniczącego – OpenTelemetry Collector – który odpowiada za ich odbiór, przetwarzania i przekierowanie do właściwych systemów docelowych. Zgodnie z konfiguracją projektu, metryki są eksportowane do Prometheus, ślady do Jaeger, a dziennik zdarzeń do systemu Loki za pośrednictwem protokołu OTLP HTTP. Architektura warstwy obserwowalności systemu została szerzej omówiona w podrozdziale 5.7.

4.18. SeaweedFS

SeaweedFS jest rozproszonym systemem składowania plików o otwartym kodzie źródłowym (ang. *open-source*), zaprojektowanym z myślą o wydajnej obsłudze dużej liczby plików przy zachowaniu stałego czasu dostępu do danych [37]. Przechowywanie plików binarnych bezpośrednio w relacyjnej bazie danych jest rozwiązaniem powszechnie odradzonym, które prowadzi do szybkiego wzrostu bazy, pogorszenia wydajności zapytań oraz utrudnienia tworzenia kopii zapasowych. SeaweedFS jest rozwiązaniem tego problemu jako dedykowany system składowania obiektów.

Ważną właściwością architektury systemu jest sposób zarządzania metadanymi. W przeciwieństwie do rozwiązań, gdzie przechowuje mapowanie wszystkich plików, SeaweedFS deleguje metadane poszczególnych plików do serwerów woluminów, a węzeł główny zarządza jedynie informacjami o woluminach. Dzięki temu odczyt pliku wymaga co do zasady tylko jednej operacji dyskowej, co klasyfikuje dostęp jako $O(1)$. Jedną z istotnych zalet systemu jest kompatybilność z interfejsem API Amazon S3, pozwalającym na integrację z dowolnym oprogramowaniem obsługującym ten standard bez konieczności modyfikacji kodu klienta.

W ramach projektu Glowing Waffle AMS SeaweedFS pełni rolę lokalnego składu obiektów (ang. *self-hosted object storage*), uruchamianego jako kontener Docker z włączonym interfejsem S3. Mikroservis Incident Service wykorzystuje go do przechowywania załączników dołączanych do zgłoszeń incydentów – zdjęć uszkodzeń i innych plików dokumentacyjnych zdarzenie. Komunikacja z

SeaweedFS odbywa się przy użyciu standardowego klienta S3, skonfigurowanego zmiennymi środowiskowymi wskazującymi adres końcówki oraz dane uwierzytelniające.

4.19.Prometheus

Prometheus jest otwartym systemem monitorowania i alarmowania, pierwotnie stworzonym w serwisie SoundCloud w 2012 roku. Od tego czasu projekt zyskał szerokie grono użytkowników i w 2016 roku dołączył do Cloud Native Computing Foundation jako drugi, po Kubernetes, projekt pod jej patronatem [38].

Podstawowym modelem działania Prometheusa jest aktywne pobieranie metryk (ang. *scraping*) z monitorowanych celów poprzez protokół HTTP, w regularnych odstępach czasu. Każda zebrana wartość przechowywana jest jako szereg czasowy (ang. *time series*), czyli ciąg pomiarów opatrzonych w znacznikiem czasu i zestawem etykiet opisujących ich kontekst. Do odpytywania zebranych danych służy dedykowany język zapytań PromQL, umożliwiający filtrowanie, agregowanie i analizowanie metryk w czasie rzeczywistym. Prometheus projektowy jest z myślą o niezależności każdego węzła, co oznacza, że serwer nie wymaga zewnętrznego składowania danych ani innych zdalnych usług, co skutkuje jego odpornością na awarie pozostałych komponentów infrastruktury [38].

W systemie Glowing Waffle AMS Prometheus zbiera metryki eksportowane przez OpenTelemetry Collector, który pośredniczy między mikroserwisami a systemem monitorowania. Konfiguracja projektu definiuje dwa cele zbierania danych: instancję OpenTelemetry Collector eksponującą metryki aplikacyjne oraz broker RabbitMQ, udostępniający własne metryki operacyjne na dedykowanym porcie. Interwał pobierania metryk wynosi dziesięć sekund, a zebrane dane są następnie wizualizowane w narzędziu Grafana, stanowiącym warstwę prezentacyjną dla całej infrastruktury monitorowania, co szczegółowo opisano w podrozdziale 5.7.

4.20.Grafana

Grafana jest otwartą platformą do wizualizacji i analizy danych, umożliwiającą odpytywanie, prezentację i eksplorację metryk, dzienników zdarzeń oraz śladów niezależnie od miejsca ich przechowywania [39]. Jej kluczową zaletą jest niezależność od źródła danych, ponieważ Grafana nie przechowuje danych samodzielnie, lecz pełni rolę warstwy prezentacyjnej integrującej się z różnorodnymi systemami backendowymi poprzez system wtyczek źródłowych (ang. *data sources*). Pozwala to na tworzenie ujednoliconych pulpitów nawigacyjnych (ang. *dashboards*) agregujących dane z wielu systemów w jednym miejscu.

Centralnym elementem interfejsu platformy jest panel, czyli wizualizacja danych w wybranej formie graficznej, np. wykresu liniowego, wykresu słupkowego lub tabeli. Panele są grupowane w pulpity nawigacyjne, które można parametryzować zmiennymi, filtrować zakres czasowy danych oraz udostępniać je innym użytkownikom. Grafana oferuje również wbudowany mechanizm alarmowania, pozwalający na definiowanie reguł opartych na zapytaniach do źródeł danych i wysyłaniu powiadomień do skonfigurowanych kanałów.

System Glowing Waffle AMS wykorzystuje Grafane jako centralny interfejs warstwy obserwowaności. Skonfigurowano trzy źródła danych odpowiadające trzem rodzajom sygnałów telemetrycznych: Prometheus jako źródło metryk, Jaeger jako źródło śladów rozproszonych oraz Grafana Loki jako źródło danych dzienników zdarzeń. Każde z tych źródeł jest skonfigurowane jako wtyczka w Grafanie, która odpytuje je bezpośrednio i prezentuje pobrane dane w postaci paneli i pulpitów nawigacyjnych. Zestawienie tych systemów w ramach jednego interfejsu umożliwia korelowanie danych telemetrycznych z różnych źródeł, co zostało szerzej omówione w podrozdziale 5.7.

4.21. Jaeger

Jaeger jest platformą o otwartym dostępie służącą do śledzenia rozproszonego (ang. *distributed tracing*), pierwotnie opracowaną przez firmę Uber Technologies w 2016 roku i przekazaną pod patronat Cloud Native Computing Foundation, gdzie uzyskała status projektu ukończonego (ang. *graduated project*) [40]. Śledzenie rozproszone pozwala na rejestrowanie i wizualizację pełnej ścieżki żądania przechodzącego przez wiele serwisów w systemie rozproszonym, co jest szczególnie istotne przy diagnozowaniu problemów wydajnościowych i identyfikowaniu wąskich gardeł.

Podstawowym pojęciem w modelu danych Jaeger jest ślad (ang. *trace*), który stanowi reprezentację pełnej ścieżki żądania w systemie, złożoną w jednostkowych odcinkach (ang. *spans*). Każdy odcinek opisuje pojedynczą operację wykonywaną w ramach żądania, zawierając informacje o czasie rozpoczęcia, czasie trwania, nazwie serwisu oraz dodatkowych atrybutach kontekstowych. Odcinki są ze sobą powiązane hierarchicznie, tworząc strukturę drzewa przyczynowo-skutkowego. Jaeger jest w pełni kompatybilny ze standardem OpenTelemetry, przyjmując dane telemetryczne w protokole OTLP, co czyni go oczywistym elementem stosu obserwowalności opartego na tym standardzie [40].

W systemie Glowing Waffle AMS Jaeger uruchamiany jest w trybie all-in-one, łączącym wszystkie komponenty w jednym kontenerze Docker – jest to rozwiązanie uproszczające konfigurację przy zachowaniu pełnej funkcjonalności. Ślady generowane przez mikroserwisy są przesyłane do Jaegera za pośrednictwem OpenTelemetry Collectora, który pełni rolę pośrednika między źródłami danych a systemami docelowymi. Zebrane ślady są następnie wizualizowane w Grafanie, skonfigurowanej jako klient platformy, co opisano w podrozdziale 4.20.

4.22. Grafana Loki

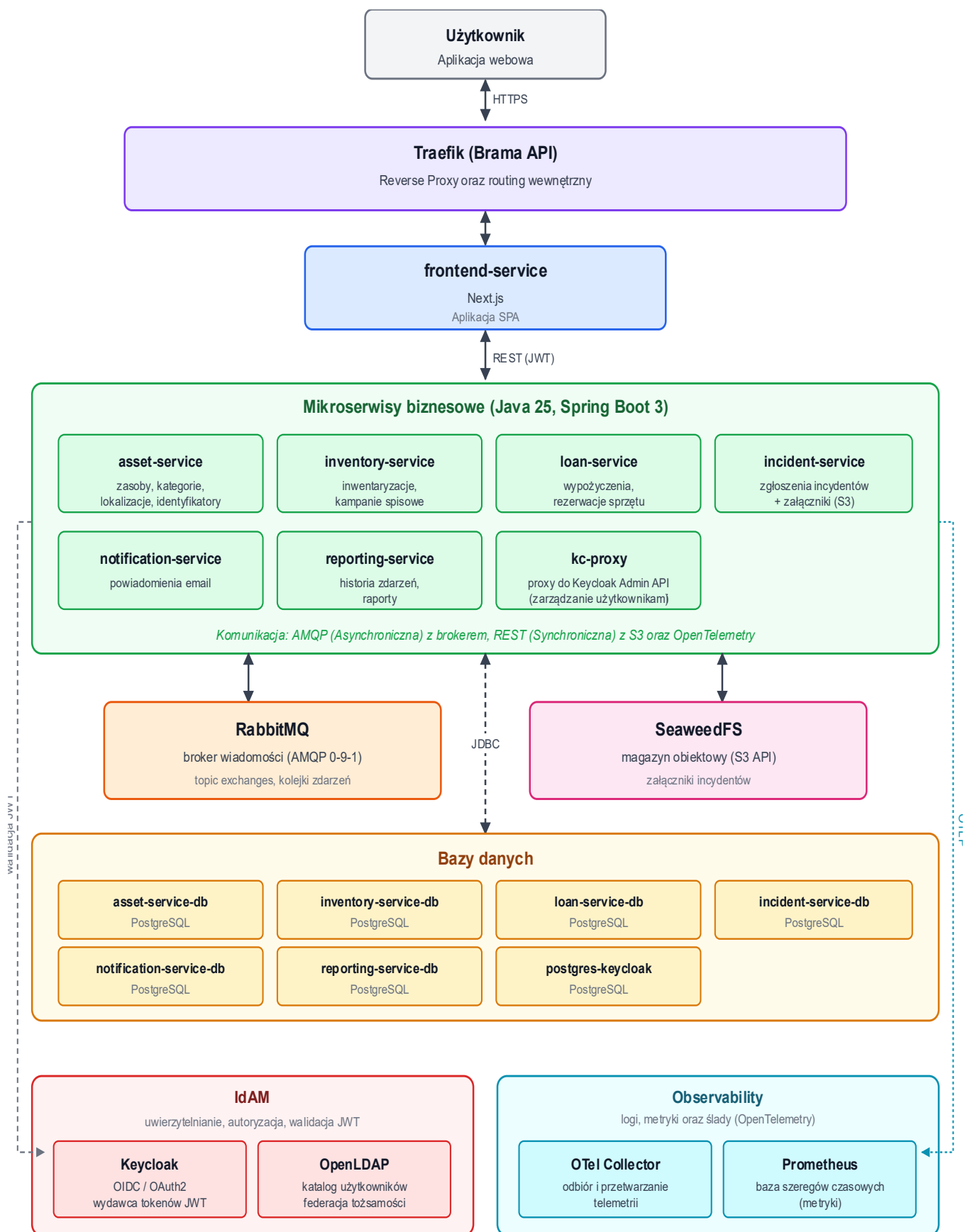
Grafana Loki jest horyzontalnie skalowalnym systemem agregacji dzienników zdarzeń (ang. *log aggregation*), tworzonym przez Grafana Labs i inspirowanym podejściem stosowanym w systemie Prometheus [41]. Jego wyróżniającą cechą na tle innych rozwiązań tego typu jest rezygnacja z indeksowania treści dzienników, a zamiast tego Loki indeksuje wyłącznie metadane w postaci etykiet (ang. *labels*) opisujących strumień dzienników, takich jak nazwa serwisu czy środowisko uruchomieniowe. Samo wyszukiwanie pełnotekstowe w treści dzienników jest nadal możliwe, jednak etykiety służą do efektywnego zawężania zbioru danych przed jego przeszukiwaniem. Takie podejście znacząco obniża koszty składowania i przetwarzania dzienników przy zachowaniu funkcjonalności przeszukiwania.

Do odpytywania danych zgromadzonych w Loki służy dedykowany język zapytań LogQL, wzorowany składniowo na PromQL stosowanym w Prometheusie. Pozwala on na filtrowanie strumieni dzienników według etykiet, wyszukiwanie wyrażeń regularnych w treści wpisów oraz tworzenie zapytań metrycznych, które agregują dane z dzienników w postaci szeregów czasowych.

Przedstawiony projekt wykorzystuje system Grafana Loki jako centralny system gromadzenia dzienników zdarzeń ze wszystkich mikroserwisów. Wpisy są przesyłane do Loki za pośrednictwem OpenTelemetry Collectora przy pomocy protokołu OTLP HTTP, co eliminuje konieczność instalowania dedykowanego agenta zbierającego dzienniki. Zebrane dane są następnie wizualizowane za pośrednictwem systemu Grafana, skonfigurowanego jako klient Loki, co pozwala na ich eksplorację w tym samym interfejsie co metryki i ślady rozproszone.

5. Decyzje architektoniczne

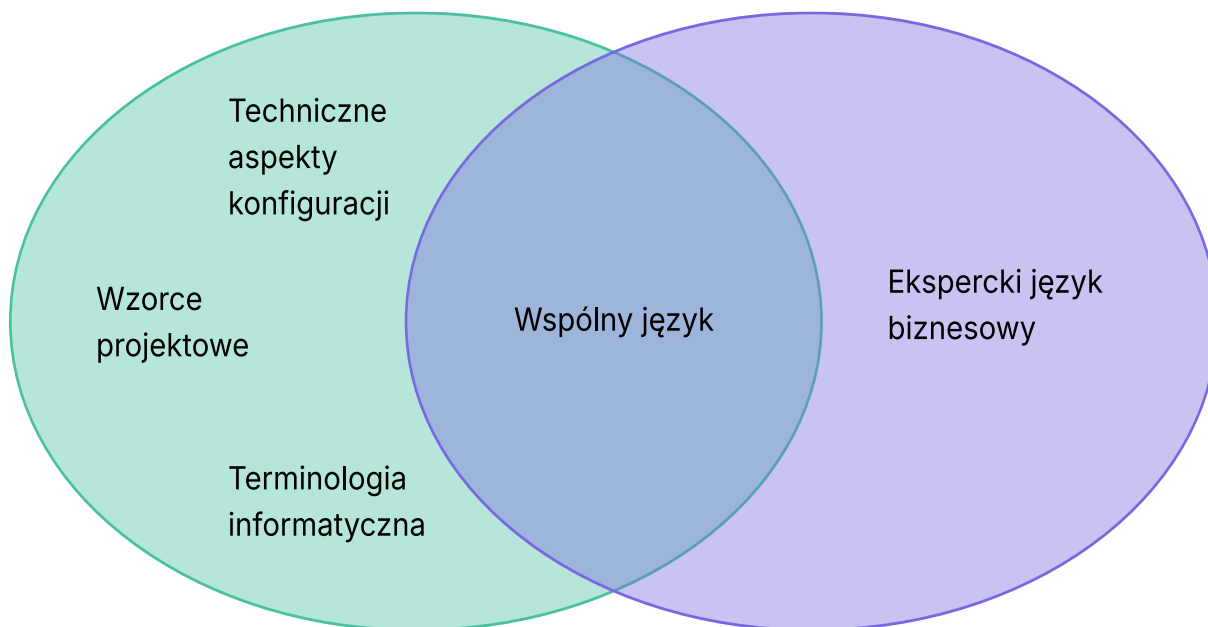
Mark Richards i Neal Ford w [42] (str. 17) twierdzą, że „wszystko w architekturze oprogramowania jest kompromisem”. Z powyższej tezy wyprowadzić można również wniosek, iż nie ma uniwersalnie dobrych bądź złych praktyk architektonicznych, które miałyby zastosowanie w każdym projekcie. Próba stosowania jednego wzorca do każdego problemu często owocuje w systemach, które są nadzbyt złożone, kosztowne w utrzymaniu i trudne w rozwoju. Niniejszy rozdział uwypukla kluczowe decyzje architektoniczne podjęte w tym projekcie, a także przedstawia wynik świadomego rachunku kompromisów wynikających z zastosowanych rozwiązań. Wypadkową powyższych decyzji jest architektura systemu, której widok logiczny wraz z wyodrębnionymi komponentami przedstawiono na Rysunku 25. Rysunek przedstawia ogólną architekturę systemu Glowing Waffle AMS, zaprojektowanego jako aplikacja webowa oparta na podejściu mikroserwisowym. Warstwa dostępową wykorzystuje Traefik jako bramę API, natomiast interfejs użytkownika oparty jest na frameworku Next.js. Komunikacja frontendu z backendem odbywa się przy pomocy REST API z uwierzytelnieniem opartym na JWT i OAuth 2.0. Poszczególne mikroserwisy odpowiadają za wydzielone obszary domenowe systemu, korzystając z niezależnych baz danych PostgreSQL, komunikacji asynchronicznej przez brokera wiadomości RabbitMQ, centralnego systemu IdAM Keycloak oraz infrastruktury obserwowalności opartej na OpenTelemetry i Prometheusie.



Rysunek 25 Architektura systemu Glowing Waffle AMS. Źródło: Opracowanie własne.

5.1. Uzasadnienie wyboru architektury mikroserwisowej

Wybór stylu architektonicznego jest jedną z najważniejszych decyzji podejmowanych na etapie analizy. Determinuje on bowiem złożoność projektu na całej przestrzeni cyklu życia aplikacji. Według [42] (str. 245), architektura mikroserwisowa zdobyła swoją dzisiejszą nazwę i zyskała na popularności w 2014 roku, dzięki postowi blogowemu, który został stworzony przez Martina Fowlera oraz Jamesa Lewisa. Owi autorzy zwrócili uwagę na nowy, rozproszony styl architektoniczny, składający się z małych i niezależnych od siebie części – zwanych mikroserwisami. Części te czerpały z istniejącego już wtedy podejścia do tworzenia oprogramowania zwanego *DDD – Domain-driven design*, czyli projektowania zorientowanego na dziedzinę. Według Evansa [43] (str. 68-74), lwią część złożoności aplikacji nie wynika z aspektu technicznego, lecz z dziedziny biznesowej, dlatego model dziedziny biznesowej powinien stanowić centralny element architektury systemu. Podejście to wprowadza w użycie pojęcie ograniczonego kontekstu (ang. *Bounded Context*), które wyznacza granice modelu dziedziny w zakresie jego spójności i jednoznaczności; każdy kontekst posiada własny, zwany przez Evansa „wspólny język” (ang. *Ubiquitous Language*), stanowiący iloczyn zbiorów między technicznymi aspektami projektu, językiem domenowym oraz biznesowym językiem eksperckim. Diagram wspólnego języka jest przedstawiony na Rysunku 26.

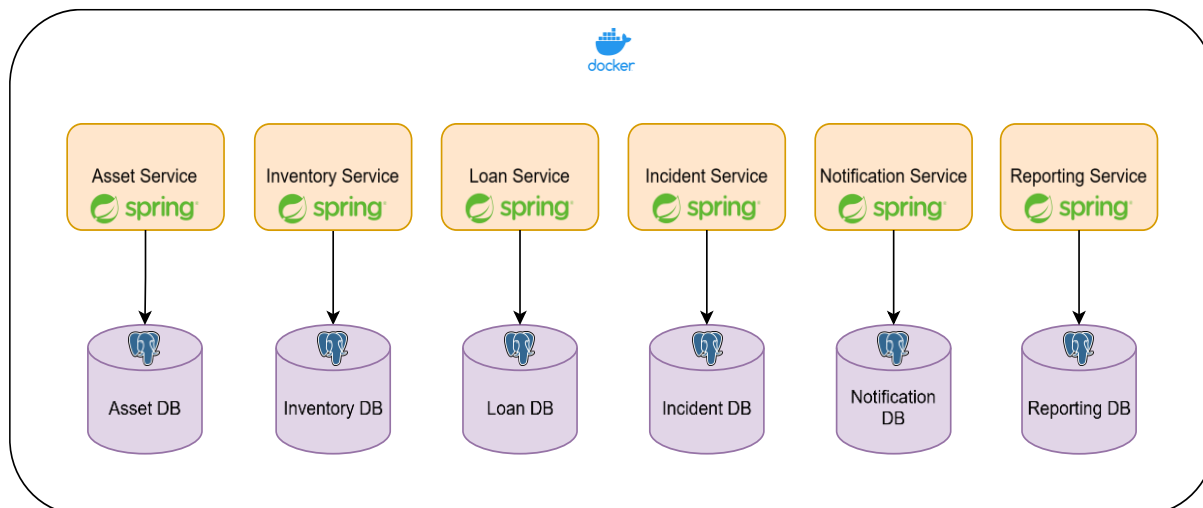


Rysunek 26 Wspólny język na diagramie Venna. Źródło: Opracowanie własne.

Opisane przez Evansa pojęcie ograniczonego kontekstu stało się wyjściowym elementem dekompozycji w architekturze mikroserwisowej. W systemie Glowing Waffle Asset Management System wyodrębniono sześć takich kontekstów, a każdy z nich zmaterializował się do postaci osobnego mikroserwisu:

- Asset Service – odpowiadający za cykl życia środka trwałego w firmie.
- Inventory Service – zarządzający inwentaryzacjami magazynowymi.
- Incident Service – rejestrujący zgłoszenia problemów ze środkami w systemie.
- Loan Service – umożliwiający śledzenie przypisania i wypożyczeń środków.
- Notification Service – wyłącznie wysyłający powiadomienia do użytkowników systemu.
- Reporting Service – pełniący rolę mikroserwisu zbierającego zdarzenia audytowe od innych mikroserwisów w systemie.

Każdy z mikroservisów posiada własny model dziedziny oraz wspólny język. Według [44] (str. 12), cechą wyróżniającą architekturę mikroservisową jest luźne powiązanie (ang. *loose coupling*). Oznacza to, że mikroservisy powinny komunikować się tylko przy użyciu własnych API, które bezpośrednio enkapsulują logikę biznesową, bez możliwości ingerencji w bazę danych drugiego serwisu. System Glowing Waffle AMS implementuje ten wzorzec architektoniczny („*Database per service*”) przez zapewnienie każdemu mikroservisowi osobnej instancji bazy danych, co przedstawiono na Rysunku 27.



Rysunek 27 Realizacja wzorca "Database per service". Źródło: Opracowanie własne.

Luźne powiązanie serwisów i ich niezależność ma niebagatelny wpływ na szybkość procesu wytwarzania oprogramowania. W systemie Glowing Waffle AMS, każdy mikroservis posiada własny obraz Docker oraz dedykowany potok *CI/CD*. Dzięki takiemu rozwiązaniu, zmiana w logice w jednym mikroservisie nie wymaga zmian w drugim. Zmiany mogą być wykonywane przez wiele zespołów w jednym czasie. Dodatkowo, zmiana może być wdrożona i przetestowana bez przebudowy całego środowiska produkcyjnego. Według [45], organizacja Atlassian zmieniła częstotliwość wdrożeń produkcyjnych z jednego tygodniowo do dwóch lub trzech dziennie dzięki wyeliminowaniu ciężkiej architektury monolitowej, a skrócenie cyklu wydawania oprogramowania jest podawane jako jedno z pierwszych i wymiernych korzyści, bezpośrednio odczuwanych przez zespoły programistyczne.

Według [44], (str. 30-31) dojrzałe firmy korzystające z potoków *CI/CD* i mikroservisów, są rzadziej obciążone długimi oknami niedostępności aplikacji; są w stanie także szybciej przywrócić usługi w przypadku dłuższej awarii. Niezależność wdrożeń i rozłączność mikroservisów przekłada się również na odporność systemu, postrzeganego jako szersza usługa biznesowa, na awarię. Przykładowo, serwis Notification Service odpowiada wyłącznie za dostarczanie powiadomień drogą e-mail – jego ewentualna niedostępność skutkuje przerwaniem ciągłości notyfikacji, nie zaś problemami z przeglądaniem środków bądź dokonywaniem inwentaryzacji. Granica kontekstu jest jednocześnie granicą propagacji awarii. W architekturze monolitycznej taka izolacja nie jest możliwa bez zastosowania dodatkowych mechanizmów ochronnych na poziomie kodu.

Sukcesywną zaletą dekompozycji jest również możliwość niezależnego skalowania serwisów, stosownie do ich rzeczywistego poziomu obciążenia. W tradycyjnej architekturze monolitowej, celem obsłużenia zwiększonego ruchu na jednym module, należy zwiększyć zasoby dostępne dla całej aplikacji (często po prostu dla całej maszyny wirtualnej), co prowadzi do marnotrawstwa zasobów i wzrostu kosztów. W systemie Glowing Waffle AMS przewidywany jest asynchroniczny przyrost ruchu; podczas gdy Asset Service oraz Loan Service będą najbardziej obciążone w ciągu godzin biznesowych przedsiębiorstwa, Notification Service i Incident Service będą cechowały się inną charakterystyką ruchu. Według [42] (str. 264), architektura mikroservisowa cechuje się jako styl architektoniczny,

najwyższą elastycznością i skalowalnością spośród powszechnie używanych, co w kontekście zmiennych profili obciążenia przekłada się na oszczędności zasobów przedsiębiorstwa.

Uczciwy rachunek kompromisów wymaga jednak zestawienia wyżej wymienionych korzyści z wadami i kosztami, które niesie ze sobą architektura mikroserwisowa. Przede wszystkim należy tutaj wymienić utratę atomowości operacji, które rozciągają się na więcej niż jeden serwis. Standardowo, architektura monolityczna zapewnia spójność przez mechanizm dwufazowego zatwierdzania *2PC* (ang. *Two-Phase Commit*), w ramach jednej bazy danych. Oznacza to, że transakcja albo jest w całości zatwierdzana, albo w całości wycofywana. Własność ta, będąca częścią szerszego zestawu gwarancji *ACID* (ang. *Atomicity, Consistency, Isolation, Durability*), jest fundamentem niezawodności relacyjnych baz danych. W architekturze mikroserwisowej globalny protokół *2PC* nie jest możliwy do zaimplementowania w sposób niezawodny. Protokół ten bowiem oczekuje blokowania zasobów na czas głosowania przez wszystkich uczestników, co w przypadku awarii jednego z mikroserwisów spowoduje paraliż innych procesów. Do czasu rozwiązania konfliktu system pozostawałby w stanie niespójnym. Oprócz tego, istotną wadą *2PC* w kontekście systemów rozproszonych jest fakt wprowadzania silnego sprzężenia czasowego, co stoi w jawnej sprzeczności z ideą luźnego powiązania, będącej fundamentem architektonicznym tego podejścia.

W odpowiedzi na wyżej wymieniony problem, zaproponowany został wzorzec Sagi, czyli sekwencji lokalnych, atomowych transakcji, z których każda publikuje zdarzenie inicjujące następny krok danej transakcji. W przypadku błędu w którymkolwiek z kroków, dokonywana jest tak zwana operacja kompensująca (ang. *Compensating transaction*), która przywraca stan system sprzed awarii. Według [44] (str. 117-125), wyróżnione są dwa główne warianty implementacji wzorca Sagi. Pierwszym z nich jest Saga oparta na choreografii. Wariant ten polega na autonomicznej reakcji mikroserwisów na zdarzenia opublikowane przez inne serwisy, bez udziału centralnego koordynatora. Drugą opcją implementacji wyżej wymienionego wzorca jest Saga oparta na orkiestracji. W tym podejściu logika koordynacji procesu biznesowego oddelegowana jest do komponentu zwanego orkiestratorem, który pełni rolę dyrygenta. Orkiestrator wysyła wiadomości do uczestników transakcji rozproszonej, oczekuje na ich odpowiedź, a następnie podejmuje decyzję o dalszych krokach (tj. kontynuacja procesu bądź wykonanie transakcji kompensujących).

W systemie Glowing Waffle AMS zdecydowano się na zastosowanie sagi opartej na choreografii. O wyborze przesądziła stosunkowo niewielka liczba kroków w ramach jednej transakcji biznesowej; bowiem większość operacji domenowych zamykała się w kontekście jednego mikroserwisu. W takim wypadku, narzut technologiczny wynikający z utrzymania dodatkowego orkiestratora przewyższałby korzyści z centralizacji przepływu. Komunikacja międzymikroserwisowa zorientowana na zdarzenia została zaimplementowana przy pomocy brokera wiadomości RabbitMQ, co szczegółowo omówiono w podrozdziale 5.2.

Istotną wadą architektury mikroserwisowej jest narzut sieciowy wynikający z konieczności komunikacji się ze sobą różnych mikroserwisów, co wprowadza nieuchronne ryzyko błędów i opóźnień. Według [44] (str. 124-126), sieć w systemach rozproszonych powinna być traktowana jako zasób zawodny, z tendencją do bycia niedostępną, z opóźnieniem oraz ograniczeniami przepustowości. Oznacza to, że każde wywołanie sieciowe musi jasno definiować kroki podejmowane w przypadku niepowodzenia żądania; standardowo stosowanymi rozwiązaniami są tutaj limity czasowe żądań synchronicznych, ponawianie z wycofaniem wykładniczym (ang. *Exponential backoff*) oraz wzorzec bezpiecznika (ang. *Circuit breaker*).

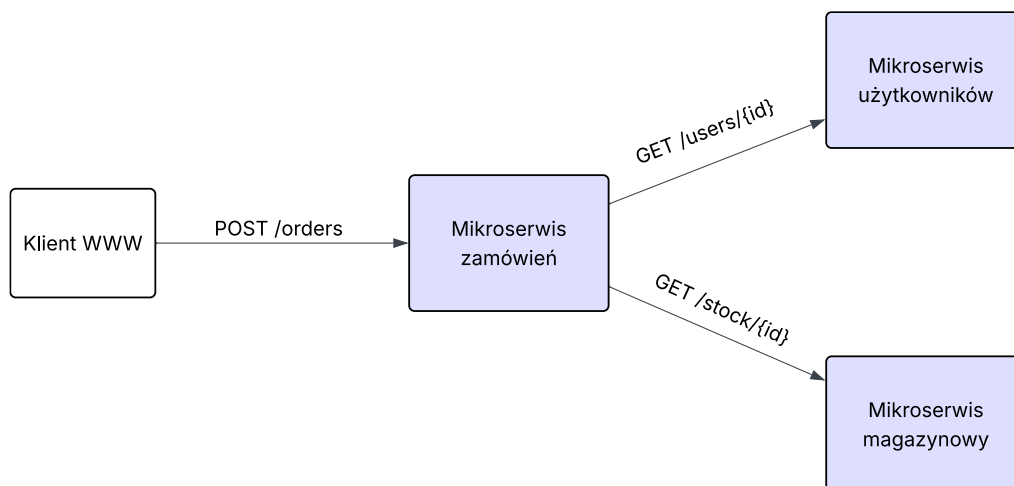
Trzecim obszarem dodatkowych kosztów jest złożoność operacyjna i obserwowalność systemu. W architekturze monolitycznej cykl życia żądania można łatwo prześledzić poprzez analizę pojedynczego pliku dziennikowego (tj. logu aplikacji). W systemie rozproszonym jedno żądanie generuje zazwyczaj kaskadę zdarzeń obejmującą wiele komponentów (mikroserwisy, *API Gateway*, instancje baz danych, broker wiadomości itd.), z których każdy wymaga osobnego profilowania i obserwowania. Zrozumienie i diagnoza problemu w takim ekosystemie wymaga zaawansowanych mechanizmów obserwowalności, które będą w stanie złączyć wiele elementów transakcji w jedną całość

– poprzez agregację logów, metryk oraz śledzenia rozproszonego (ang. *Distributed tracing*). W systemie Glowing Waffle AMS warstwa obserwowalności została oparta na powszechnym standardzie OpenTelemetry oraz systemie metryk Prometheus, których konfigurację oraz szersze uzasadnienie omówiono w podrozdziale 5.7.

Analizując wszystkie wady i zalety tego stylu architektonicznego, należy podkreślić, że w kontekście prototypu zastosowanie architektury mikroserwisowej wygenerowało dużą złożoność operacyjną, która w rzeczywistym systemie produkcyjnym powinna być jasno uzasadniona skalą zespołu oraz wymaganiami niefunkcjonalnymi. W przeciwnym wypadku dojść może do sytuacji, która w języku angielskim otrzymała miano *overengineering* – „przekombinowania” bądź „nadmiernej inżynierii” – a zatem sytuacji, gdzie zaproponowane rozwiązanie drastycznie odstaje od realnych wymagań biznesowych, jest zbyt skomplikowane jak na dziedzinę problemową i nie przynosi korzyści.

5.2. Komunikacja asynchroniczna przy użyciu brokera wiadomości

Według [44] (str. 103-105), komunikacja synchroniczna jest jednym ze sposobów realizacji komunikacji międzyprocesowej (ang. *Interprocess Communication*), i zazwyczaj prowadzi do obniżonej dostępności aplikacji. Problem ten wynika z dwóch istotnych kwestii. Pierwszą z nich jest fakt, że w modelu synchronicznym główne żądanie zostaje zablokowane do czasu otrzymania wszystkich podrzędnych odpowiedzi, które są wymagane do poprawnego przeprocesowania tego żądania. Problem ten zilustrowany został na przykładowej architekturze w Rysunku 28.



Rysunek 28 Blokowanie głównego żądania przez żądania pochodne. Źródło: Opracowanie własne.

W powyższym przykładowym modelu, główne żądanie użytkownika do końcówki „/orders” zostaje zablokowane do czasu, aż mikroserwis zamówień dostanie odpowiedź na wywołanie dwóch końcówek w innych mikroserwisach – „/users/{id}” oraz „/stock/{id}”. Jeżeli system biznesowy realizuje zadane żądanie jako sekwencje synchronicznych wywołań między n serwisami, to całkowita dostępność usługi zdefiniowana jest jako iloczyn dostępności każdego z serwisów. Oznacza to, że niska dostępność jednego serwisu rzutuje negatywnie na cały system, co w systemach dedykowanych przedsiębiorstwom nie jest sytuacją akceptowalną.

Drugą istotną wadą, którą również można dostrzec na Rysunku 3, jest fakt, iż protokół HTTP (będący de facto standardem w architekturze REST) nie ma wbudowanego mechanizmu obsługi sytuacji niedostępnych końcówek. W przypadku braku odpowiedzi mikroserwisu, klient otrzymuje błąd na poziomie warstwy transportowej (np. *Connection Refused*, *Timeout*), bądź kod HTTP 5xx, jeżeli pośrednik zwraca błąd zastępczy. Komunikat, o ile nie został wcześniej utrwalony po stronie nadawcy,

zostaje bezpowrotnie utracony. Odpowiedzialność za retransmisję żądania spada na nadawcę; na jego barki również pozostaje buforowanie żądań i implementacja mechanizmu wycofania wykładniczego.

Według [44] (str. 104), w architekturze rozproszonej dobrą praktyką jest maksymalizacja użycia komunikacji asynchronicznej. W związku z tym, a także celem zaadresowania powyższych problemów, w systemie Glowing Waffle AMS przyjęto komunikację asynchroniczną przy pomocy brokera jako podstawowy mechanizm komunikacji między mikroserwisami.

W modelu opartym na pośredniku wiadomości, takim jak RabbitMQ, problemy pojawiające się w komunikacji synchronicznej rozwiązywane są w sposób natywny. Komunikat publikowany przez nadawcę/producenta (ang. *Producer*) nie jest kierowany bezpośrednio do odbiorcy/konsumenta (ang. *Consumer*), lecz przekazywany do pośredniczącego komponentu – brokera – bezpośrednio odpowiedzialnego za jej trwałe przechowywanie, kolejkovanie oraz dostarczenie co najmniej jeden raz do konsumenta bądź konsumentów. [44] (str. 93) definiuje następujące, ogólne zalety korzystania z brokera komunikatów:

- Luźne powiązanie – nadawca tworzący notyfikację nie musi znać szczegółów lokalizacji serwisu, do którego wysła wiadomość.
- Buforowanie komunikatów – broker sam zarządza mechanizmami buforowania i przeciwdziałania przeciążeniu konsumenta. Dodatkowo, niezależnia dostępność producenta od konsumenta; konsument może być niedostępny w czasie tworzenia komunikatu, podłączyć się do brokera i z opóźnieniem przetworzyć wiadomość.

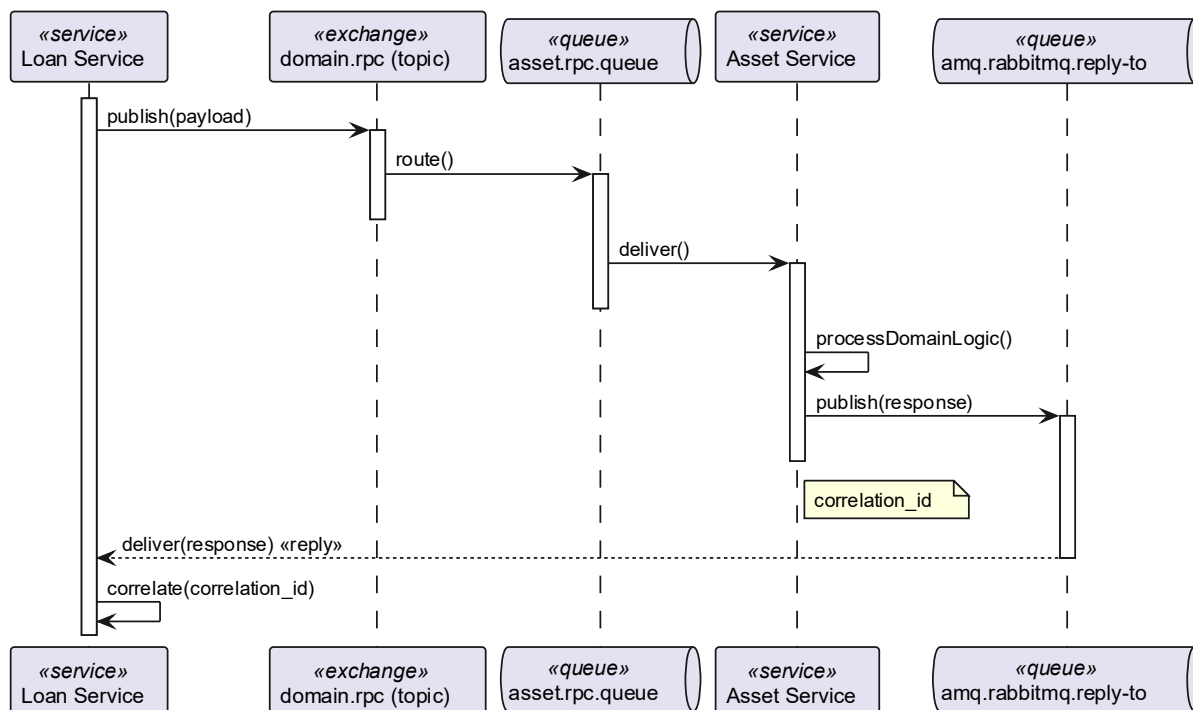
Elastyczność w wyborze stylu komunikacji – większość czołowych rozwiązań rynkowych (RabbitMQ, Apache Kafka, ActiveMQ i wiele innych) oferuje dowolność wyboru, spośród różnych rodzajów komunikacji:

- zdalne wywołanie procedury (ang. *Remote Procedure Call*)
- komunikat jednokierunkowy (powiadomienie bez oczekiwania na odpowiedź) dla jednego bądź wielu konsumentów
- strumieniowanie

Spośród dostępnych technologii brokerów wiadomości, zdecydowano się na użycie RabbitMQ, implementującego protokół *AMQP* (*Advanced Message Queuing Protocol*). Rozwiązanie to jest zoptymalizowane pod kątem elastyczności routingu i prostoty w implementacji. Profil komunikacyjny systemu Glowing Waffle AMS obejmuje zdarzenia domenowe o stosunkowo niewielkim wolumenie i wysokiej różnorodności typów. Przemawia to na korzyść RabbitMQ dla celów tej pracy, względem konkurencyjnego rozwiązania Apache Kafka, dedykowanego środowiskom do przetwarzania strumieniowego o ogromnej skali. Innymi argumentami przemawiającymi za tym wyborem była dobra integracja z ekosystemem frameworka Spring przy pomocy biblioteki Spring *AMQP* oraz wbudowana obsługa niedostarczonych wiadomości za pomocą kolejki *DLQ* (ang. *Dead Letter Queue*).

W systemie Glowing Waffle AMS wyodrębniono dwa logiczne kanały komunikacji asynchronicznej, zrealizowane jako dwa wymienniki typu *topic*. Pierwszy z nich, *domain.events*, służy do publikacji informacji o zdarzeniach dokonanych w systemie, takich jak zmiana statusu środka, zakończenie kampanii inwentaryzacyjnej czy też utworzenie incydentu.

Drugi wymiennik, *domain.rpc*, używany jest do realizacji asynchronicznych zapytań w stylu *RPC*. Wykorzystywany jest w sytuacjach, gdy jeden mikroserwis musi pozyskać dane z drugiego. Diagram sekwencji dla takiego przepływu został przedstawiony na Rysunku 29. W przypadku wywołań typu *RPC*, klient tworzy tymczasową kolejkę „*reply-to*”, przeznaczoną tylko na poczet odpowiedzi do żądania.



Rysunek 29 Diagram sekwencji komunikacji w stylu RPC w brokerze RabbitMQ. Źródło: Opracowanie własne.

Istotnym aspektem implementacyjnym jest moment publikacji komunikatów do brokera. W systemie Glowing Waffle AMS przyjęto regułę, zgodnie z którą żadne zdarzenie nie zostaje wysłane przed zatwierdzeniem lokalnej transakcji bazodanowej. Bez takiego zabezpieczenia istniałoby ryzyko wysłania wiadomości o zdarzeniu, które w rzeczywistości zostało wycofane w bazie danych, co w konsekwencji doprowadziłoby do desynchronizacji stanów między mikroservisami. Celem zwiększenia niezawodności systemu, zastosowano również mechanizm kolejek niedostarczonych wiadomości (*DLQ*). Dzięki temu, komunikat, który po określonej liczbie prób nie zostaje pomyślnie odebrany przez konsumenta, jest przekierowywany do kolejki *DLQ*, przez co nie blokuje przetwarzania dalszych komunikatów ani nie wpada w nieskończoną pętlę.

Jak zaznaczono we wstępie do rozdziału 5, każda decyzja jest swego rodzaju kompromisem. Nie inaczej jest z implementacją brokera wiadomości.[44] (str. 94) uwypukla następujące wady i trudności, jakie się z tym wiążą:

- Dodatkowa złożoność operacyjna – broker sam w sobie jest kolejną aplikacją, którą należy wdrożyć, konserwować i utrzymywać w środowisku. Zwiększa to nakład pracy w zespole.
- Pojedynczy punkt awarii – jeżeli broker nie obsługuje bądź nie został wdrożony w trybie wysokiej dostępności *HA* (ang. *High Availability*), jego awaria może doprowadzić do niedostępności całego systemu przez awarię pojedynczego punktu, zwaną *SPoF* (ang. *Single Point of Failure*).
- Ryzyko wydajnościowe – jeżeli broker nie będzie w stanie odpowiednio buforować i przysyłać komunikatów do klientów, istnieje ryzyko narzutu na cały system.

W przypadku wybrania RabbitMQ, istotna jest konieczność projektowania idempotentnych konsumentów; według [22], standard *AMQP* składa obietnicę gwarancji dostarczenia pojedynczej notyfikacji co najmniej jeden raz (ang. *at-least-once delivery*). Oznacza to, że wiadomość może zostać dostarczona do konsumenta wielokrotnie, co bez mechanizmu idempotencji może skutkować w błędnej operacji bądź niechcianym zwielokrotnieniu danych.

5.3. Komunikacja synchroniczna REST

W rozdziale 5.2 jako podstawowy mechanizm komunikacji międzymikroserwisowej wskazano model asynchroniczny oparty na brokerze wiadomości RabbitMQ. Mimo to, w systemie Glowing Waffle AMS nie ma możliwości całkowitego wyeliminowania komunikacji synchronicznej. Aplikacja frontendowa, zaprojektowana we frameworku Next.js, z natury obsługuje cykl żądanie-odpowiedź protokołu HTTP, celem zwrócenia danych do ich prezentacji. Próba zastąpienia wyżej wymienionej metody komunikacji alternatywami asynchronicznymi (takimi jak np. *WebSockets*, czyli technologia zapewniająca ciągłą dwukierunkową komunikację przez połączenie TCP), skutkowałaby rozwiązaniem nieproporcjonalnie złożonym do skali problemu; dodatkowo, system Glowing Waffle AMS nie jest systemem opartym o strumieniowanie danych, tj. nie przesyła do klientów danych krytycznych czasu rzeczywistego – na przykład zmieniających się kursów walut, wiadomości czatu, terminali do zawierania transakcji *CFD* (ang. *Contract for difference*), aktualizacji dokumentów od innych uczestników i tym podobnych.

Podjęto decyzję, że wszystkie interakcje synchroniczne będą oparte na stylu architektonicznym REST, pierwotnie opisanym przez Roya Fieldinga w jego rozprawie doktorskiej. Według [46], jest to zbiór ograniczeń nakładanych na system rozproszony. Najistotniejszą z nich, mającą realne przełożenie na decyzje projektowe dla prototypu, to bezstanowość interakcji (ang. *Statelessness*), umożliwiająca skalowanie poziome instancji bez obaw o replikację danych sesyjnych. Rozważono przy tym dwie alternatywne technologie – pierwszą z nich jest *gRPC*, będący protokołem binarnym opartym na standardzie HTTP/2. Naturalnym środowiskiem tego są jednak aplikacje strumieniujące czasu rzeczywistego w interakcjach serwer-serwer, których przykłady były wymienione wyżej. Dodatkową trudnością implementacyjną owego protokołu od strony przeglądarki jest konieczność korzystania z dodatkowej warstwy pośredniczącej (*gRPC-Web*), co zaprzecza idei prostoty komunikacji klient-serwer. Drugą z rozważanych alternatyw był *GraphQL*; według [44] (str. 281), jest to nowoczesna i popularna technologia bazująca na grafach. Modeluje dane przechowywane na serwerze w formie grafu, a klientowi udostępnia końcówkę, która akceptuje skomponowane żądania, i zwraca tylko te dane, o które klient zapytał. Niestety, z wprowadzeniem tej technologii do systemu musi wiązać się bardzo szczegółowa analiza celem wykluczenia zapytań typu N+1. Istotnym ograniczeniem tej technologii jest również konieczność wprowadzenia globalnego schematu danych, co stałoby w sprzeczności z ideą dekompozycji i ograniczonego kontekstu, którą to szerzej omówiono w rozdziale 5.1.

Świadomą decyzją architektoniczną było ograniczenie obecności REST wyłącznie do granicy systemu, tj. do interakcji pomiędzy aplikacją kliencką a backendem traktowanym jako całość. Mikroserwisy nie komunikują się między sobą synchronicznie przy pomocy HTTP; wszelkie interakcje wewnętrzne są realizowane przez asynchroniczną wymianę komunikatów na brokerze RabbitMQ, omówioną w podrozdziale 5.2. Dopuszczenie synchronicznych wywołań REST między mikroserwisami doprowadziłoby do problemu iloczynu dostępności, opisanego w podrozdziale 5.2, oraz zniweczyłoby wszelkie korzyści wynikające z wbudowanych mechanizmów buforowania wiadomości przez brokera. Inną, ale równie istotną wadą komunikacji synchronicznej to wprowadzenie tak zwanego sprzężenia czasowego (ang. *temporal coupling*) – protokół HTTP wymaga bowiem obu stron komunikacji, klienta i serwera, do poprawnego działania w tym samym czasie.

Na brzegu systemu zastosowano również wzorzec *API Gateway*. Według [44] (str. 259-265), jest to usługa zapewniająca obsługę wejścia do architektury systemu, będąca rozwiązaniem problemu wielu klientów żądających dostępu do wielu mikroserwisów, których adres IP jest zawsze tymczasowy (restart środowiska kontenerowego powoduje rotację adresów). Brama pełni rolę pojedynczego punktu wejścia odpowiedzialnego za routing żądań na podstawie reguł URL a dodatkowo stanowi naturalne miejsce wymuszenia uwierzytelnienia żądań. Rozwiązanie typu *API Gateway* często oferuje szereg innych usług, takich jak równoważenie obciążenia dla mikroserwisów (ang. *load balancing*), automatyczne odkrywanie usług w klastrze Docker oraz Kubernetes oraz wdrożeń bezprzerwowych typu *canary*. W systemie Glowing Waffle AMS zdecydowano się na użycie darmowego, otwarto źródłowego API Gateway, jakim jest Traefik. Zastosowano wzorzec segregacji ruchu *north-south* (tj.

ruchu przychodzącego z Internetu), oraz *east-west* (tj. komunikacja wewnątrz klastrowa, między mikroserwisami, np. wysłanie wiadomości do brokera, a także z warstwy frontendowej). Niestety, Traefik w wersji darmowej nie wspiera weryfikacji tokenów JWT na poziomie żądania, w związku z czym odbywa się to na poziomie każdego mikroserwisu przy pomocy biblioteki Spring Security oraz Keycloak, pełniącego rolę serwera autentykacji i autoryzacji. Traefika jako technologię szczegółowo omówiono w podrozdziale 4.10.

W kontekście wyboru konkretnej technologii API Gateway rozważono trzy warianty: Spring Cloud Gateway, bezpośrednio zintegrowany z ekosystemem Spring, klasyczny webserwer Nginx z manualnie definiowanymi regułami routingu oraz Traefik, integrujący się natywnie z platformą konteneryzacji. Wybór padł na Traefik z trzech zasadniczych powodów. Po pierwsze, API Gateway pozostaje niezależne od stosu technologicznego mikroserwisów – odróżnia to tę bramę od wersji Springowej, w której ewentualna wymiana frameworka aplikacyjnego mikroserwisów na inny język wymagałaby równoczesnej wymiany całego rozwiązania bramy. Po drugie, Traefik realizuje konfigurację routingu w sposób deklaratywny, na podstawie etykiet (*ang. labels*) kontenerów Docker, co eliminuje konieczność utrzymywania osobnego pliku konfiguracyjnego synchronizowanego ze stanem środowiska uruchomieniowego. Po trzecie, brama ta natywnie eksportuje metryki oraz ślady w standardzie OpenTelemetry, stanowiącym podstawę warstwy obserwowalności omawianej w podrozdziale 5.7. Co do Nginx, dynamiczne dostosowywanie jego konfiguracji do zmiennej liczby instancji mikroserwisów wymagałoby dodatkowych komponentów pomocniczych, co wprowadzałoby niepotrzebną i nieuzasadnioną złożoność, gdyż inne rozwiązania są w stanie robić to natywnie.

Jak każda decyzja architektoniczna, również wybór REST jako kontraktu z aplikacją kliencką oraz wzorca API Gateway niesie ze sobą pewne ograniczenia. Po pierwsze, synchroniczna natura interakcji sprawia, że niedostępność któregośkolwiek z mikroserwisów manifestuje się natychmiast po stronie klienta; w środowisku produkcyjnym wymagałoby to zastosowania mechanizmów odporności, takich jak wzorzec bezpiecznika (*ang. Circuit Breaker*) bądź ponawiania z wycofaniem wykładniczym. W prototypie mechanizmy te zostały świadomie pominięte i odnotowane jako kierunek dalszego rozwoju. Po drugie, sama brama API pozostaje pojedynczym punktem awarii (*SPoF*) – jej niedostępność uniemożliwia jakąkolwiek interakcję klienta z systemem. W środowisku produkcyjnym należałoby zatem wdrożyć Traefik w trybie wysokiej dostępności z wieloma replikami za zewnętrznym równoważeniem obciążenia. Po trzecie, REST jako styl architektoniczny nie narzuca szczegółów implementacji operacji na domenę, co czyni go elastycznym, ale wymaga zarazem wewnętrznej dyscypliny zespołu w utrzymaniu spójności konwencji – każdy nowo dodawany zasób musi respektować ustalone reguły, w przeciwnym wypadku ograniczenie jednolitego interfejsu ulega wypaczeniu wraz z upływem czasu.

5.4. Serwer autoryzacji i autentykacji

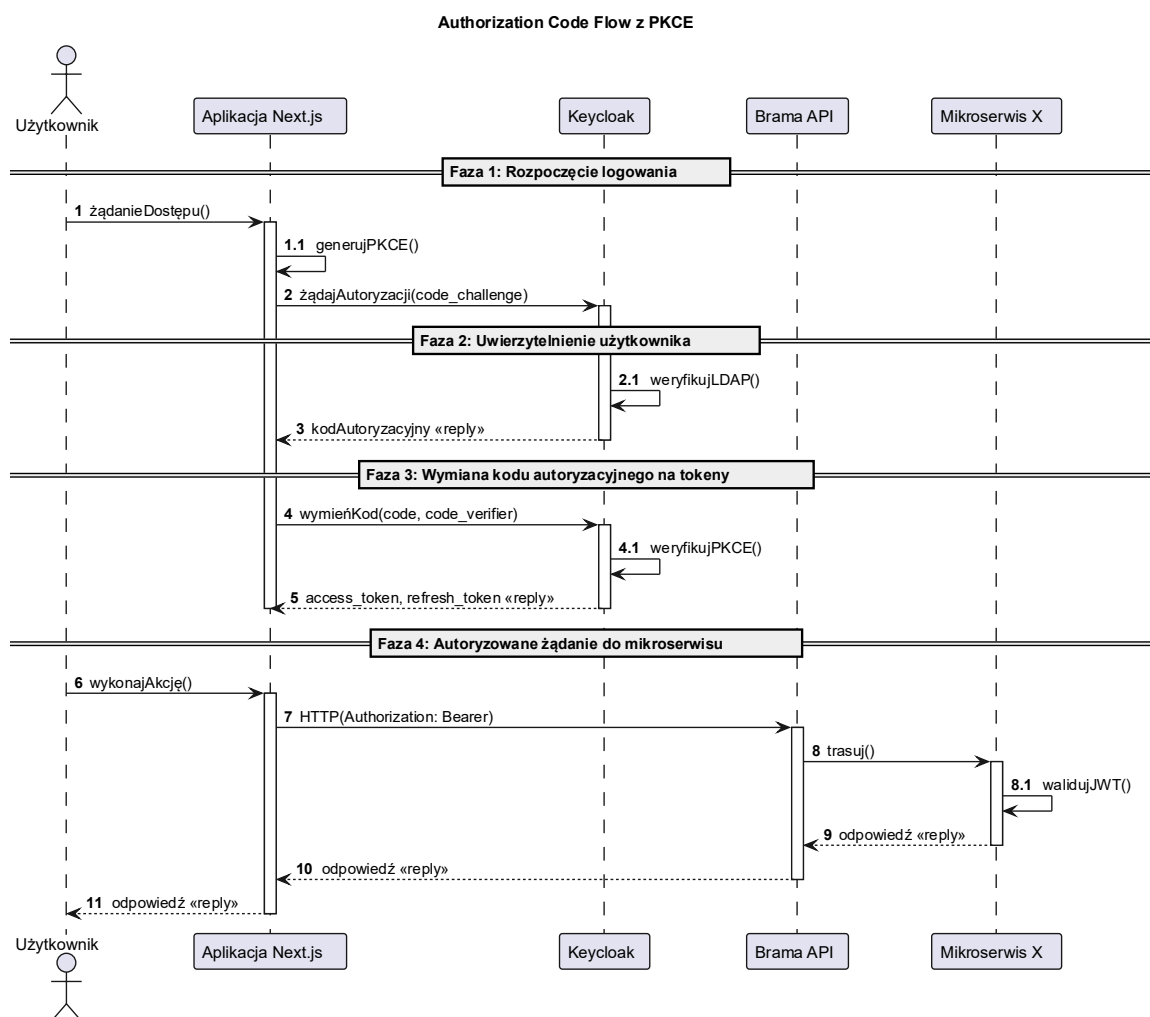
System Glowing Waffle AMS od początku był projektowany z myślą o byciu wdrożonym w organizacjach co najmniej średnich, mających istniejącą architekturę informatyczną z katalogiem użytkowników w pewnym centralnym systemie typu LDAP bądź podobnym (np. Microsoft Active Directory). Z tego względu podjęto decyzję o nieprzechowywaniu użytkowników i ról bezpośrednio w aplikacji, gdyż doprowadziłoby to w pewnym sensie do redundancji danych – użytkownicy już istniejący w strukturze domenowej firmy musieliby zostać odtworzeni w serwisie, a konta i role wymagałyby utrzymywania w wielu miejscach. Dodatkowo, rotacja pracowników (np. zwolnienia) wymusza na administratorach dodatkową pracę. Jeżeli użytkownikami zarządza się z poziomu centralnego, łatwiej wymusić określone polityki firmowe (np. zmiana hasła co 90 dni, maksymalna długość sesji, siła hasła itd.), a także zapewnić zgodność z regulacjami dotyczącymi danych osobowych (np. RODO).

W systemie Glowing Waffle AMS rolę centralnego serwera autoryzacji i uwierzytelnienia powierzono otwartoźródłowemu rozwiązaniu Keycloak, którego szerszą charakterystykę przybliżono w

podrozdziale 4.11. W tym miejscu skupiono się wyłącznie na decyzjach architektonicznych i implementacyjnych, które wynikają z zastosowania tego narzędzia oraz jego integracji z ekosystemem Spring. Kluczowym bodźcem do wyboru tego serwera była natywna implementacja dwóch standardów branżowych, czyli OAuth 2.0 oraz OpenID Connect (OIDC). Pierwszy z nich, zgodnie ze standardem [18] ; Microsoft, „Type Inference,” Microsoft, [Online]. Available: <https://www.typescriptlang.org/docs/handbook/type-inference.html>. [Data uzyskania dostępu: 19 05 2026]., stanowi protokół delegowania uprawnień dostępu, natomiast drugi jest warstwą tożsamości zbudowaną nad OAuth 2.0, dodającą mechanizm uwierzytelniania użytkownika oraz format tokenu identyfikacyjnego (ang. *ID Token*). Oparcie się o standardowe praktyki zmniejsza ryzyko uzależnienia się od jednego dostawcy (ang. *vendor lock-in*), oraz umożliwia migracje na inne serwery autoryzacji w przyszłości (np. Authentik, Ory Hydra bądź inne rozwiązanie typu *Identity & Access Management*) jeżeli w danym przedsiębiorstwie zaszłaby taka potrzeba.

W standardzie OAuth 2.0 istnieje kilka modeli autentykacji. Po analizie, zdecydowano się na przyjęcie wzorca *Authorization Code Flow* z rozszerzeniem *PKCE* (*Proof Key for Code Exchange*). W kontekście systemu Glowing Waffle AMS przepływ ten przebiega następująco, co zilustrowano na Rysunku 30:

- Użytkownik, próbując zalogować się i uzyskać dostęp do aplikacji klienckiej (Next.js), jest przekierowywany do strony logowania udostępnianej przez Keycloak. Przed wykonaniem przekierowania aplikacja kliencka generuje losowy ciąg weryfikacyjny (ang. *code verifier*) oraz jego skrót kryptograficzny (ang. *code challenge*), który dołącza do żądania autoryzacyjnego.
- Po pomyślnym uwierzytelnieniu użytkownika przez Keycloak (przy użyciu poświadczeń z LDAP), następuje przekierowanie przeglądarki z powrotem do aplikacji klienckiej, dołączając do adresu URL jednorazowy kod autoryzacyjny.
- Następnie aplikacja kliencka, w wywołaniu serwer-serwer, wymienia ten kod oraz pierwotny ciąg weryfikacyjny na zestaw tokenów: token dostępu, token odświeżający oraz token identyfikacyjny. Keycloak weryfikuje, czy skrót ciągu weryfikacyjnego odpowiada wcześniej zarejestrowanemu wyzwaniu, co stanowi istotę zabezpieczenia PKCE.
- Uzyskany token dostępu jest następnie dołączany do każdego żądania kierowanego do API Gateway, która rozprowadza ruch do właściwych mikroservisów, gdzie podlega on lokalnej weryfikacji przez bibliotekę Spring Security, co opisano w dalszej części podrozdziału. Wybór wariantu z PKCE jest podyktowany specyfiką aplikacji klienckiej, która z perspektywy serwera autoryzacji klasyfikowana jest jako klient publiczny (ang. *public client*), niezdolny do bezpiecznego przechowywania sekretu klienta (ang. *client secret*).

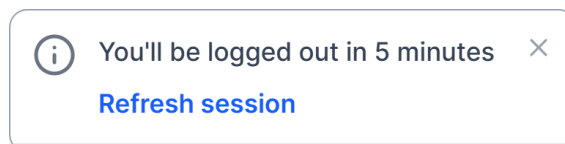


Rysunek 30 Diagram sekwencji autoryzacji typu Authorization Code Flow z rozszerzeniem PKCE. Źródło: Opracowanie własne.

Mechanizm *PKCE* skutecznie zabezpiecza przepływ przed atakiem przechwycenia kodu autoryzacyjnego (ang. *authorization code interception attack*).

Tokeny dostępu (ang. *access tokens*) wydawane przez Keycloak zaimplementowano w formacie JSON Web Token (JWT). JWT to samodzielny, podpisany cyfrowo nośnik informacji o uwierzytelnionym użytkowniku, składający się z trzech części: nagłówka, ładunku (ang. *payload*) oraz podpisu. Najistotniejszą zaletą tego formatu w kontekście architektury mikroserwisowej jest jego bezstanowość – mikroserwis weryfikujący żądanie nie musi komunikować się z serwerem autoryzacji w celu walidacji tokenu, lecz wykonuje to lokalnie, weryfikując podpis przy pomocy klucza publicznego pobranego z końcówki JWKS (*JSON Web Key Set*) Keycloaka. Rozwiązanie to wpisuje się w opisaną w podrozdziale 5.1 ideę luźnego powiązania komponentów oraz odporności na awarie. Chwilowa niedostępność serwera autoryzacji nie powoduje paraliżu systemu, ponieważ wcześniej wydane tokeny pozostają ważne do momentu wygaśnięcia. Czas życia tokenu dostępu ustawiono na jedną godzinę, co stanowi kompromis pomiędzy bezpieczeństwem (krótki czas życia ogranicza okno czasowe na nadużycie – co szerzej omówiono w ostatnim akapicie tego podrozdziału) a wydajnością (zbyt częste odświeżanie generowałoby nadmierne obciążenie serwera autoryzacji); z punktu widzenia biznesowego, system przechowuje informacje będące tajemnicą przedsiębiorstwa, takie jak dane faktur oraz ceny środków trwałych. Wraz z tokenem dostępu wydawany jest token odświeżający (ang. *refresh token*) o znacznie dłuższym czasie życia, używany przez klienta do uzyskania nowego tokenu dostępu

bez konieczności ponownego logowania użytkownika; to rozwiązanie zostało zaimplementowane po stronie frontendu jako wyskakujący dymek z przyciskiem, który umożliwia bieżącemu użytkownikowi przedłużenie tokenu jednym kliknięciem. Zostało to przedstawione na Rysunku 31.



Rysunek 31 Dymek typu pop-up informujący użytkownika o konieczności przedłużenia sesji. Źródło: Opracowanie własne.

Model autoryzacji w systemie Glowing Waffle AMS oparto na rolach, przypisywanych na poziomie usługi katalogowej LDAP. Każdy uwierzytelniony użytkownik otrzymuje w tokenie JWT listę przypisanych ról, które są następnie wykorzystywane przez mechanizmy biblioteki Spring Security do egzekwowania reguł autoryzacyjnych na poziomie poszczególnych końcówek REST. Szczegółowy opis ról systemowych zawarto w podrozdziale 3.4. Decyzja o zastosowaniu modelu RBAC (*Role Based Access Control*), zamiast bardziej zaawansowanego modelu opartego na atrybutach, czyli ABAC (*ang. Attribute-Based Access Control*), wynikała z charakterystyki domeny biznesowej. Reguły dostępu w systemie zarządzania środkami trwałymi opierają się głównie na funkcji organizacyjnej użytkownika, a nie na dynamicznych atrybutach kontekstowych żądania (np. lokalizacja, pora dnia, stanowisko, status urządzenia itp.). Wprowadzenie modelu ABAC dla potrzeb prototypu byłoby zatem przykładem wspomnianego w podrozdziale 5.1 zjawiska *overengineeringu*.

Integracja Keycloak z mikroserwisami Spring została zrealizowana w sposób jednolity przy użyciu biblioteki Spring Security i modułu oauth2-resource-server. Każdy mikroserwis pełni rolę tzw. serwera zasobów (*ang. Resource Server*), czyli komponentu, który nie bierze udziału w procesie uwierzytelniania użytkownika, lecz wymaga okazania ważnego tokenu JWT przy próbie dostępu do końcówek. Konfiguracja takiego rozwiązania od strony biblioteki jest trywialna i sprowadza się do zdefiniowania końcówki wydawcy tokenów (*ang. Issuer URI*), z której Spring Security sam pobiera klucze JWKS oraz metadane OIDC. Dzięki temu rozwiązaniu mikroserwisy nie muszą znać żadnych sekretów Keycloak (bowiem końcówka zwracająca dane JWKS jest publiczna), a sama logika weryfikacji tokenów pozostaje całkowicie ukryta za standardowymi filtrami biblioteki. Konfiguracja Keycloak dla potrzeb systemu obejmuje utworzenie dedykowanej dziedziny (*ang. realm*) o nazwie glowing-waffle, w której zdefiniowano klienta aplikacyjnego dla frontendu oraz mapowania ról systemowych.

Wybór scentralizowanego rozwiązania IdAM ma niewątpliwie wiele zalet, lecz nie jest całkowicie pozbawione wad. Po pierwsze, komponent ten zwiększa złożoność operacyjną, wymaga utrzymania własnej bazy danych, regularnego monitorowania i konserwacji. Po drugie, mimo iż chwilowa niedostępność tego systemu zazwyczaj nie stanowi problemu (z powodu przetrzymywania kluczy przez aplikację), to jego trwała awaria uniemożliwi rozpoczynanie nowych sesji oraz przedłużanie już istniejących. Oznacza to zatem kolejny punkt pojedynczej awarii (*SPoF*) mogący sparaliżować cały system; w środowisku produkcyjnym należy wdrożyć Keycloak w trybie wysokiej dostępności (*HA*) celem minimalizacji tego ryzyka.

Istotną wadą tokenów JWT jest również brak natywnego mechanizmu ich unieważniania; raz wydany token jest ważny do czasu wygaśnięcia. Otwiera to okno do wielu nadużyć:

- Token, który został przejęty w sposób nieuprawniony, może być użyty w sposób nieautoryzowany.
- Token, który został „unieważniony” w systemie IdAM – z powodu braku mechanizmu propagacji tej informacji do systemu końcowego, np. serwera zasobów – nadal pozostaje ważny i może być wykorzystywany.

Druga z wymienionych wad tokenu JWT wymaga szczególnej uwagi w systemach, które z natury obsługują dane wrażliwe (medyczne, wojskowe, finansowe itd.), gdzie w momencie zabrania uprawnień użytkownikowi poświadczeń oczekuje się natychmiastowego odcięcia dostępu. W takim środowisku należałoby zastosować wzorzec introspekcji (ang. *Token Introspection*), który polega na wykonaniu żądania do serwera autoryzacji, za każdym razem, gdy klient próbuje użyć danego tokenu JWT, celem jego sprawdzenia. Niestety, introspekcja tokenu wiąże się z utratą najważniejszej cechy, która wyróżnia technologię JWT – czyli bezstanowość.

5.5. Warstwa bazodanowa

W systemie Glowing Waffle AMS zdecydowano się na zastosowanie wzorca *Database Per Service*, opisanego po krótko w rozdziale 5.1 Według [44] (str. 42-43), każdy mikroservis musi posiadać wyłączny dostęp do własnej instancji bazy danych, a jakakolwiek wymiana informacji między aplikacjami opiera się wyłącznie przez zdefiniowane interfejsy; innymi słowy, niedopuszczalna jest bezpośrednia modyfikacja danych danego serwisu przez inny mikroservis. Gdyby ten wzorzec został zignorowany, to naruszyłoby zasadę luźnego powiązania, która stanowi fundament architektury mikroservisowej.

W prototypie zaimplementowano wzorzec izolacji na poziomie osobnych instancji silnika bazodanowego, uruchamianych jako odrębne kontenery Docker. Tym samym warto wspomnieć, że taki typ izolacji nie jest jedynym możliwym. Autor [44] wymienia również dwie następujące alternatywy:

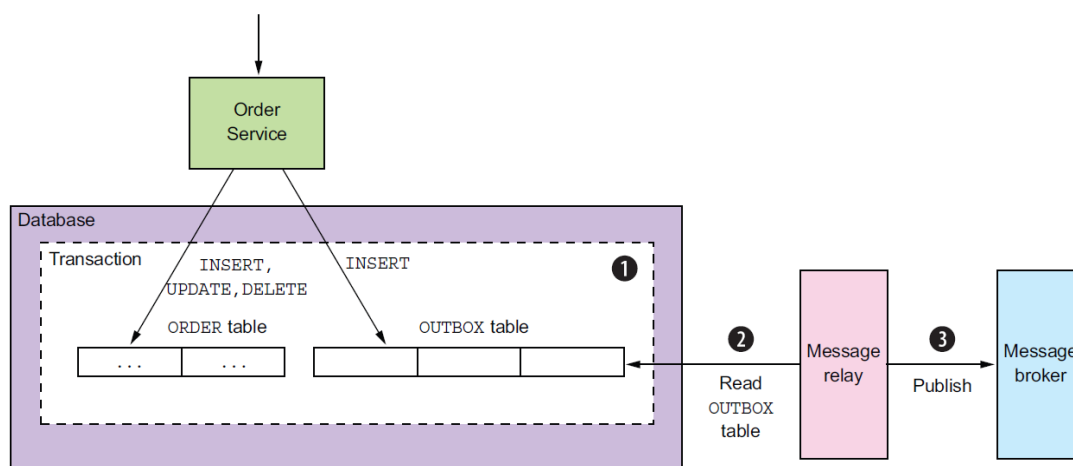
- Tabele per mikroservis – w tym rozwiązaniu, każdy mikroservis jest właścicielem swoich tabel i tylko na nich ma uprawnienia do zapisu i odczytu. Instancja bazy danych pozostaje ta sama.
- Schemat per mikroservis – schemat jako kolekcja pewnych obiektów bazodanowych (tabel, procedur, indeksów itd.), jest w posiadaniu (ang. *schema owner*) przez odpowiedni mikroservis, i posiada uprawnienia tylko na nich. W tym przypadku instancja bazy danych również jest jedna.

Obie z wyżej wymienionych opcji mają swoje niewątpliwe zalety; jedną z nich jest zmniejszona złożoność zarządzania i mniejszy narzut zasobowy (zużycie procesora, pamięci RAM, dysku itd.). Niestety, takie rozwiązanie wprowadza pojedynczy punkt awarii dla całego systemu. Mając na uwadze te wady, zdecydowano się na pełną separację na poziomie procesu, akceptując wyższy narzut pamięciowy w zamian za pełną niezależność.

Jako silnik bazodanowy zdecydowano się na otwartoźródłowy system PostgreSQL w wersji 17, który przybliżono w podrozdziale 4.5. Decyzja o wyborze relacyjnego modelu, zamiast alternatywy nierelacyjnych (np. MongoDB, Cassandra oraz wiele innych) podyktowana była realiami biznesowymi, w jakich osadzona jest aplikacja – środek trwały z natury posiada pewne asocjacje, takie jak określona kategoria, typ, przypisany użytkownik i tym podobne. Dodatkowym i istotnym argumentem jest natywne wsparcie dla typu danych *jsonb* przez PostgreSQL, pozwalająca na przechowywanie elastycznych struktur danych.

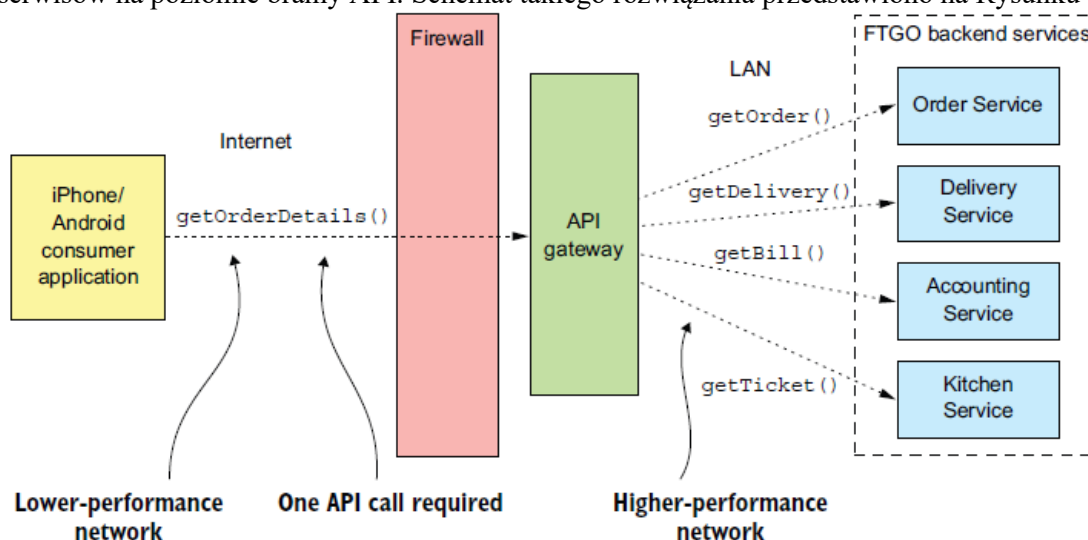
Zarządzanie transakcjami w obrębie pojedynczego mikroservisów zrealizowano przy pomocy adnotacji `@Transactional` dostarczanej przez framework Spring. Każda operacja modyfikująca stan bazy opakowana jest w transakcję ACID, gwarantującą atomowość zmian w obrębie jednej bazy danych. Jak omówiono w podrozdziale 5.1, w architekturze rozproszonej nie istnieje natywny mechanizm globalnej transakcji obejmującej wiele baz, dlatego operacje rozciągające się na więcej niż jeden mikroservis realizowane są poprzez wzorzec Sagi opartej na choreografii. Z tym wiąże się problem podwójnej operacji zapisu (ang. *dual write problem*) – sytuacji, w której proces aplikacyjny musi w atomowy sposób zapisać zmianę do bazy oraz wysłać notyfikację do brokera; w przypadku awarii pomiędzy tymi krokami jeden z zapisów może zostać utracony, a drugi nie, prowadząc do desynchronizacji. W systemie Glowing Waffle AMS zaadresowano go przez przyjęcie reguły publikacji zdarzeń wyłącznie po pomyślnym zatwierdzeniu transakcji lokalnej. Według [44] (str. 97-98), pełniejszym rozwiązaniem byłby wzorzec *Transactional Outbox*, polegający na utrwaleniu zdarzenia w

dedykowanej tabeli „zdarzeń do wysłania” w obrębie tej samej transakcji, która modyfikuje encję biznesową, a następnie asynchronicznym przekazaniu go do brokera przez niezależny proces. Proces ten przedstawiono na Rysunku 32. Jego implementację świadomie pominięto w prototypie ze względu na nieproporcjonalny narzut implementacyjny, jednak w docelowym systemie produkcyjnym taki wzorec powinien zostać wdrożony.



Rysunek 32 Realizacja wzorca Transactional Outbox na przykładzie serwisu zamówień. Źródło: [44]

Niezależność każdego mikroservisu w warstwie bazданowej tworzy dwie istotne wady. Pierwszą z nich jest brak możliwości wykonania zapytania łączącego dane z wielu kontekstów dziedzinowych przy pomocy klauzuli JOIN. W tradycyjnej architekturze monolitycznej zapytanie łączące informacje, np. o środkach trwałych, ich wypożyczeniach oraz historii zgłoszeń serwisowych sprowadzałoby się do pojedynczego polecenia SQL, wykonywanego w sposób natywny i zoptymalizowany przez silnik bazdanowy. W systemie Glowing Waffle AMS dane te zaś znajdują się w trzech odrębnych bazach należących do różnych mikroservisów, w związku z czym operacja taka musi zostać zrealizowana na poziomie warstwy aplikacyjnej; według [44] (str. 261), rozwiązaniem tego problemu jest wzorec *API Composition*, który polega na sklejaniu ze sobą odpowiedzi wielu różnych mikroservisów na poziomie bramy API. Schemat takiego rozwiązania przedstawiono na Rysunku 33.



Rysunek 33 Realizacja wzorca API Composition na przykładzie metody getOrderDetails(). Źródło: [44]

Drugą wadą jest redundancja danych pomiędzy mikroservisami. Część informacji o tej samej jednostce dziedzinowej jest w konsekwencji przechowywana w wielu bazach danych. Powielanie danych niesie ze sobą szereg wyzwań: zwiększa przestrzeń dyskową zajmowaną przez tę samą informację, wymusza implementację mechanizmów propagacji zmian oraz wprowadza ryzyko trwałej desynchronizacji w przypadku awarii brokera lub utraty komunikatu. W systemie Glowing Waffle AMS zdecydowano się na zaakceptowanie tych wad jako naturalnego następstwa architektury mikroservisowej, ograniczając redundancję do niezbędnego minimum. Warto uwypuklić, że w aplikacjach przeznaczonych dla przedsiębiorstw często wręcz dąży się do celowego wprowadzania redundancji. Wynika to z konieczności uniknięcia problemu wąskiego gardła, ponieważ odpytywanie innych mikroservisów przez sieć (synchronicznie za pomocą REST lub gRPC) przy każdej operacji drastycznie obniża wydajność i odporność systemu na awarie. Zamiast tego, serwisy przechowują lokalne, zduplikowane kopie danych należących do innych domen, często w uproszczonej formie – np. serwis Asset może posiadać własną tabelę z podstawowymi danymi incydentów, synchronizowaną z serwisem Incident.

5.6. Składowanie plików

System Glowing Waffle AMS, jako narzędzie wspierające zarządzanie cyklem życia środka trwałego w firmie, naturalnie musi umożliwiać przechowywanie nieustrukturyzowanych danych. W szczególności dotyczy to obrazów dokumentujących stan uszkodzenia ŚT w zgłoszeniach serwisowych. Jedno z klasycznych podejść, polegające na zapisie pliku bezpośrednio w bazie danych jako *BLOB* (*Binary Large Object*), zostało odrzucone w początkowej fazie analizy. Jego wdrożenie skutkowałoby wzrostem rozmiaru bazy danych, znacząco obciążałoby mechanizmy wytwarzania kopii zapasowych i replikacji oraz zmuszałoby silnik bazodanowy do przetwarzania ruchu, do którego nie został zoptymalizowany. Z tego względu zdecydowano się na rozdzielenie odpowiedzialności pomiędzy dwie warstwy magazynowania danych: relacyjną bazę PostgreSQL, służącą do przechowywania encji biznesowych oraz dedykowany magazyn obiektowy (ang. *object storage*), odpowiadający za trwałe składowanie plików binarnych.

Magazyn obiektów jest modelem przechowywania danych, w którym pojedyncza jednostka – obiekt – składa się z samej zawartości binarnej, zestawu metadanych oraz unikalnego identyfikatora. Obiekty grupowane są w logiczne kontenery zwane kubelkami (ang. *bucket*), które stanowią płaską przestrzeń nazw, w przeciwieństwie do hierarchicznej struktury katalogów charakterystycznej dla tradycyjnych systemów plików.

Standardem branżowym dla tego modelu jest API udostępnione przez usługę Amazon S3 (ang. *Simple Storage Service*), które z czasem stało się de facto wzorcem implementowanym przez liczne rozwiązania zarówno chmurowe, jak i instalowane lokalnie. Komunikacja z magazynem odbywa się przy użyciu protokołu HTTP, a operacje wykonywane są przez standardowe metody (PUT, GET, DELETE), co czyni go naturalnym wyborem dla środowisk rozproszonych.

W kontekście prototypu Glowing Waffle AMS rozważano dwa rozwiązania typu self-hosted, oferujące zgodność z Amazon S3 API: MinIO oraz SeaweedFS. Ostatecznie wybór padł na to drugie. MinIO, mimo iż przez wiele lat stanowił najpopularniejszą otwartoźródłową implementację S3 API, w ostatnim czasie przeszedł szereg zmian, które wpłynęły na jego ocenę. Po pierwsze, modyfikacje w modelu licencyjnym wprowadziły ograniczenia dotyczące używania – każde narzędzie wykorzystujące MinIO musi być typu open source, co nie było do zaakceptowania w pracy inżynierskiej. Po drugie, panel administracyjny w wersji społecznościowej został drastycznie zubożony, wiele kluczowych funkcji do zarządzania zostało wyniesionych do płatnej wersji dla przedsiębiorstw. SeaweedFS, dystrybuowany na licencji Apache 2.0, oferuje natomiast pełną zgodność z najczęściej używanym zestawem metod S3 API, niski narzut zasobowy oraz prostą konfigurację w środowisku kontenerowym Docker.

Proces utrwalania plików w systemie Glowing Waffle AMS został architektonicznie składa się z trzech kroków. W pierwszym z nich, klient wysyła do odpowiedniego mikroserwisu (Incident Service) żądanie zawierające metadane pliku:

- typ MIME
- oryginalną nazwę

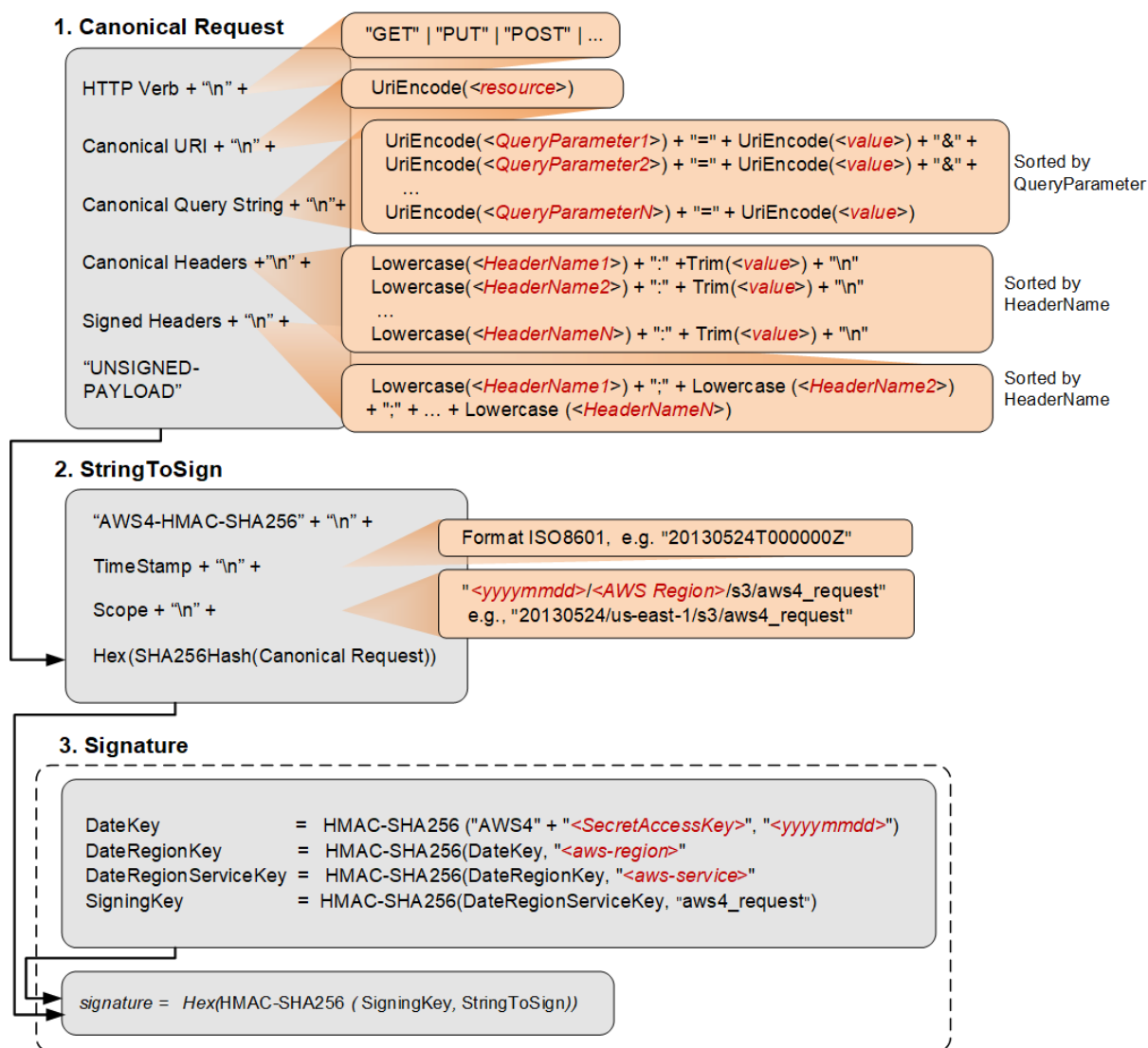
Na ich podstawie mikroserwis, korzystając z biblioteki klienckiej AWS SDK produkowanej przez Amazon, generuje podpisany kryptograficznie adres URL (ang. *pre-signed* URL), ważny przez kilkanaście minut. Adres ten zawiera wszystkie informacje niezbędne do autoryzacji żądania po stronie SeaweedFS, dzięki czemu klient może w drugim kroku wykonać żądanie typu PUT bezpośrednio do magazynu, przysyłając plik. Przykład użycia takiego żądania przedstawiono na Listingu 1, a odpowiedź na Listingu 2. Procedurę generowania kryptograficznie podpisanego URL przez bibliotekę przedstawiono na Rysunku 34 w oparciu o dokumentację Amazon.

Listing 1 Przykład pozytywnej odpowiedzi od końcówki typu *presign*, z URL potrzebnym do przesłania pliku docelowego. URL opatrzony jest wieloma parametrami służącymi do autoryzacji żądania. Źródło: Opracowanie własne.

```
curl-request POST \
--url https://incident-app.glowingwaffle.app/api/v1/incident-service/incident-
attachments/presign \
--header 'Authorization: Bearer TOKEN' \
--header 'Content-Type: application/json' \
--data '{
  "fileName" : "damaged-router.jpg",
  "contentType": "image/jpeg"
}'
```

Listing 2 Przykład pozytywnej odpowiedzi od końcówki typu *presign*, z URL potrzebnym do przesłania pliku docelowego. URL opatrzony jest wieloma parametrami służącymi do autoryzacji żądania. Źródło: Opracowanie własne.

```
{
  "uploadUrl": "https://s3-api.glowingwaffle.app/incident-attachment/31f3e85d-53e0-
46a5-8e49-bc16f586c47d.jpg?
X-Amz-Algorithm=AWS4-HMAC-SHA256
&X-Amz-Date=20260527T163045Z
&X-Amz-SignedHeaders=content-type%3Bhost
&X-Amz-Credential=7946976E18A973F31275%2F20260527%2Fus-east-1%2Fs3%2Faws4_request
&X-Amz-Expires=1500
&X-Amz-
Signature=44375cf7c2bc38d34cc743f30bfb43dc7bffd5c08986ddb9eefe8d674649efd",
  "fileKey": "31f3e85d-53e0-46a5-8e49-bc16f586c47d.jpg"
}
```



Rysunek 34 Algorytm AWS Signature Version 4 (SigV4), używany przez bibliotekę AWS SDK do generowania podpisów kryptograficznych dla nagłówków i żądań URL. Źródło: [47]

Po pomyślnym przesłaniu pliku do magazynu obiektowego, klient musi poinformować mikroserwis o takim zdarzeniu, gdyż serwis nie ma możliwości dowiedzieć się o tym w inny sposób.

Encja bazodanowa reprezentująca plik celowo nie zawiera żadnej zawartości binarnej. Przechowywane są wyłącznie metadane: typ MIME (umożliwiający aplikacji klienckiej prawidłową interpretację zawartości po pobraniu), oryginalna nazwa pliku nadana przez użytkownika (wykorzystywana wyłącznie do celów użytkowych tj. nie służy do odszukiwania pliku w magazynie obiektowym, gdyż mogłaby być niejednoznaczna i zawierać znaki problematyczne) oraz rozmiar w bajtach. Identyfikator obiektu w magazynie generowany jest po stronie serwera jako wartość UUID w wersji 4, uzupełniona o rozszerzenie pliku, co eliminuje ryzyko kolizji nazw oraz upraszcza późniejsze odszukanie zasobu. Taki podział obowiązków między dwie warstwy magazynowania danych pozostaje spójny z opisaną w podrozdziale 5.1 łącznego powiązania komponentów.

Tożsamy model został zastosowany dla operacji pobierania. W przypadku, gdy mikroserwis otrzymuje żądanie dotyczące obrazka, nie zwraca jego zawartości w odpowiedzi, lecz generuje tymczasowy, podpisany kryptograficznie adres URL, ważny przez 15 minut; podobnie jak w przypadku końcówki typu presign, URL generowany jest przez bibliotekę AWS SDK. Aplikacja kliencka w

odpowiedzi odbiera ten adres i samodzielnie pobiera plik bezpośrednio z magazynu obiektowego. Czas życia adresu stanowi kompromis pomiędzy bezpieczeństwem (krótkie okno czasowe ogranicza możliwość nadużycia w przypadku przejęcia adresu) a ergonomią użytkownika (zbyt krótkie okno wymuszałoby częste odnawianie URLów w przypadku stron prezentujących wiele obrazów). Według [47], mechanizm podpisu kryptograficznego gwarantuje co następuje:

- uniemożliwia modyfikacje żądania przechwyconego w locie, co zapobiega atakom typu MTM (ang. *man in the middle*) – każde żądanie ma wyliczony kryptograficznie skrót przy użyciu algorytmu SHA256.
- znacząco utrudnia ataki typu ponawiania żądania (ang. *replay attack*) – żądania mają bardzo krótki czas życia.

Istotną zaletą architektonicznego oparcia się o standard S3 API jest możliwość bardzo łatwej migracji prototypu na dowolne rozwiązanie chmurowe w ramach rozwoju projektu. W przypadku decyzji o wdrożeniu systemu Glowing Waffle AMS u zewnętrznego dostawcy chmurowego, zastąpienie SeaweedFS przez bardziej docelowe rozwiązanie takie jak Amazon S3, Google Cloud Storage z trybem zgodności S3 lub Cloudflare R2 sprowadza się wyłącznie do podmiany wartości w konfiguracji (adresu końcówki oraz danych uwierzytelniających). Sam kod aplikacyjny pozostaje bez zmian. Stanowi to praktyczną realizację zasady unikania uzależnienia od jednego dostawcy (ang. *vendor lock-in*), omówionej również w kontekście wyboru serwera autoryzacji w podrozdziale 5.4.

5.7. Projekt monitoringu i obserwowalności

W podrozdziale 5.1 przybliżono, że jednym z najistotniejszych kosztów architektury mikroserwisowej jest wzrost złożoności operacyjnej oraz znacznie utrudniona diagnostyka. Pojedyncze żądanie od użytkownika nie jest obsługiwane w ramach jednej aplikacji – jak by to miało miejsce w architekturze monolitowej – lecz generuje lawinę zdarzeń rozproszoną między wiele niezależnych od siebie komponentów, takich jak brama API, mikroserwisy, broker wiadomości oraz instancje bazy danych. W tak rozproszonym podejściu analiza pojedynczego pliku dziennikowego jest niewystarczająca. Z tego względu warstwa obserwowalności jest w systemie Glowing Waffle AMS pełnoprawnym komponentem, zbudowanym na obecnych standardach branżowych.

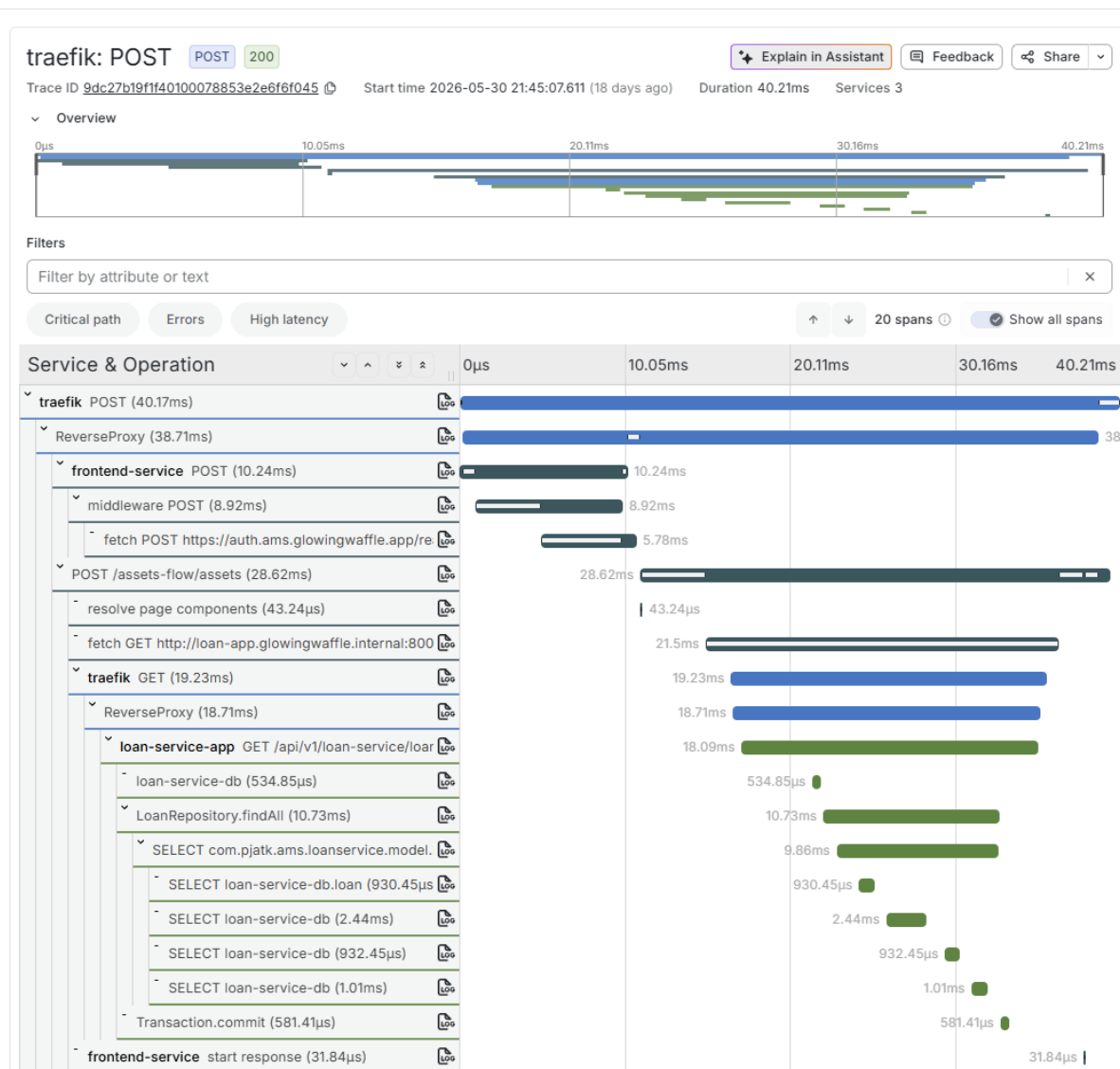
Należy w tym miejscu odróżnić dwa podobne pojęcia. Monitoring polega na obserwacji wcześniej zdefiniowanych, znanych metryk i reagowaniu na przekroczenie ustalonych progów. Jest to zatem odpowiedź na pytania, które potrafimy zadać z góry – np. monitorowanie liczby odpowiedzi zakończonych kodem 500, i odpowiedni alert, gdy ta metryka przekroczy arbitralnie zdefiniowaną liczbę. Obserwowalność jest natomiast pojęciem szerszym; według [48], określa ona stopień, w jakim na podstawie danych emitowanych przez system na zewnątrz można wnioskować o jego stanie wewnętrznym, w tym diagnozować sytuacje wcześniej nieprzewidziane. W praktyce inżynierskiej obserwowalność realizowana jest przez trzy podstawowe rodzaje danych telemetrycznych, zwane potocznie „trzema filarami obserwowalności”: dzienniki zdarzeń (ang. *logs*), metryki (ang. *metrics*) oraz ślady rozproszone (ang. *traces*).

Kluczową decyzją w tym obszarze było oparcie całej warstwy obserwowalności na otwartym standardzie OpenTelemetry (często skrótowo zapisywanym jako OTel), scharakteryzowanym w podrozdziale 4.17. OpenTelemetry to projekt prowadzony pod patronatem fundacji Cloud Native Computing Foundation (CNCF), powstały z połączenia wcześniejszych inicjatyw OpenTracing oraz OpenCensus, którego celem jest ujednolicenie sposobu generowania, zbierania i eksportu danych telemetrycznych. Motywacją do przyjęcia tego standardu w prototypie była chęć uniknięcia „wynajdywania koła na nowo” – zamiast projektować autorski, niestandardowy format telemetryczny oraz dedykowane mu narzędzia, oparto się o rozwiązanie będące de facto branżowym standardem, wspieranym natywnie przez większość współczesnych frameworków oraz platform monitorujących.

Najistotniejszą architektonicznie konsekwencją oparcia się o OpenTelemetry jest rozdzielenie warstwy instrumentacji aplikacji od warstwy składowania i wizualizacji danych (ang. *observability backend*). Kod aplikacji oraz konfiguracja instrumentacji pozostają całkowicie niezależne od konkretnego narzędzia typu APM (Application Performance Monitoring). Dzięki temu system nie uzależnia się od jednego dostawcy, a tę samą telemetrię można skierować zarówno do rozwiązania otwartoźródłowego (np. Grafana), jak i do komercyjnych platform typu SaaS, takich jak Datadog, New Relic, Dynatrace czy Grafana Cloud, bez modyfikacji kodu źródłowego samej aplikacji. Ten sposób myślenia jest tożsamy z tym przedstawionym w kontekście serwera autoryzacji Keycloak (podrozdział 5.4) oraz magazynu obiektowego zgodnego z S3 API (podrozdział 5.6); w każdym z tych przypadków oparcie się o otwarte i powszechne standardy, zamiast o zamknięte rozwiązanie konkretnego producenta, zwiększa elastyczność i przenośność systemu.

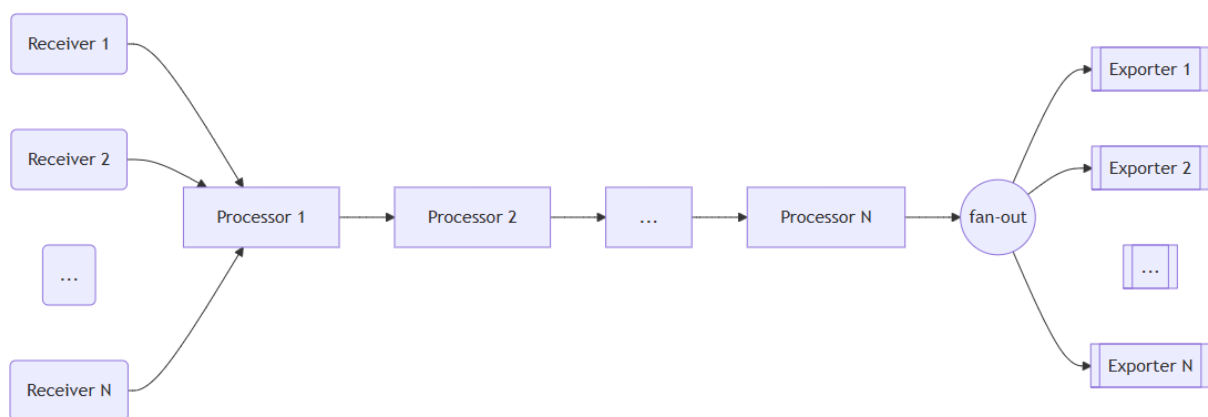
W zakresie metryk aplikacyjnych świadomie oparto się o standardowe, gotowe wskaźniki emitowane przez bibliotekę Micrometer wraz z modulem Spring Boot Actuator, zamiast definiować własne, niestandardowe pomiary. Mechanizmy te udostępniają bez dodatkowego nakładu programistycznego metryki opisujące stan Tomcata (liczba aktywnych wątków, sesji oraz zajętość puli połączeń), maszyny wirtualnej JVM (zużycie pamięci sterty, aktywność odśmieccacza pamięci oraz liczba wątków), a także charakterystykę obsługiwanych żądań HTTP (ich liczbę, rozkład czasów odpowiedzi oraz kody odpowiedzi). Zestaw ten pokrywa cztery podstawowe sygnały monitorowania (ang. *four golden signals*) – opóźnienie, ruch, błędy oraz nasycenie zasobów – które stanowią minimalny, lecz wystarczający fundament obserwacji systemu zorientowanego na obsługę użytkownika. Warto jednak zauważyć, że w docelowym systemie produkcyjnym istotne może być zdefiniowane również metryk reprezentujących domeną biznesową; w standardzie OpenTelemetry sprowadza się to do użycia SDK (ang. *Software Development Kit*) Open Telemetry i zdefiniowana logiki tworzenia nieszablonowych metryk.

Co istotne, instrumentacją objęto nie tylko mikroserwisy aplikacyjne, lecz wszystkie komponenty systemu. Brama API Traefik natywnie eksportuje ślady, metryki oraz logi w standardzie OTLP; aplikacja frontendowa Next.js została zinstrumentowana przy pomocy biblioteki zgodnej z OpenTelemetry, a komponenty infrastrukturalne, takie jak Keycloak oraz broker RabbitMQ, udostępniają własne metryki w formacie zgodnym z systemem Prometheus. Dzięki temu obraz systemu jest kompletny: obserwacją objęta jest cała ścieżka żądania, począwszy od API Gateway, mikroservisów i brokera, aż po warstwę bazodanową. Fundamentalną wartością tak zaprojektowanej warstwy jest możliwość łączenia zdarzeń pochodzących z różnych serwisów w ramach jednej logicznej transakcji biznesowej. Realizuje to mechanizm propagacji kontekstu (ang. *context propagation*). Każde żądanie wchodzące do systemu otrzymuje unikalny identyfikator (ang. *trace-id*), który jest następnie przenoszony pomiędzy kolejnymi komponentami – zarówno w nagłówkach żądań HTTP, jak i w nagłówkach komunikatów przesyłanych przez brokera RabbitMQ. W rezultacie nawet operacja asynchroniczna, przekraczająca granicę procesu i realizowana w innym mikroserwisie z opóźnieniem, pozostaje powiązana z pierwotnym żądaniem. W narzędziu klasy APM inżynier może wówczas prześledzić kompletny „ślad” pojedynczej transakcji w postaci uporządkowanego czasowo drzewa, identyfikując dokładnie ten komponent, który wprowadził błąd lub nadmierne opóźnienie. Przykładowe drzewo żądania pobierającego z systemu wypożyczenia zostało przedstawione na Rysunku 35.



Rysunek 35 Ślad zapytania pobierającego wypożyczenia w systemie Glowing Waffle AMS w stosie obserwowalności Grafana Cloud. Źródło: Opracowanie własne.

Architekturę zbierania telemetry oparto na wzorcu kolektora (OpenTelemetry Collector), pełniącego rolę centralnego pośrednika pomiędzy instrumentowanymi komponentami a docelowymi backendami. Komponenty aplikacyjne wysyłają telemetry do kolektora za pomocą protokołu OTLP, natomiast część komponentów infrastrukturalnych, takich jak RabbitMQ lub Keycloak, udostępnia metryki w formacie Prometheus na końcówce „/metrics”. W takim przypadku metryki mogą być pobierane bezpośrednio przez Prometheus albo przez kolektor skonfigurowany z odbiornikiem Prometheus. Kolektor przetwarza telemetry przy pomocy potoków (ang. *pipelines*), z których każdy składa się z odbiornika (ang. *receiver*), opcjonalnych procesorów (ang. *processor*) oraz eksportera (ang. *exporter*) [49]. Konfigurację potoków kolektora zastosowaną w prototypie, dla środowiska produkcyjnego, przedstawiono na Listingu 3. *Pipeline* produkcyjny z użyciem chmurowego stosu obserwowalności Grafana Cloud zaprezentowano na Rysunku 36.



Rysunek 36 Wizualizacja przykładowego ułożenia N procesorów danych w ramach potoku w kolektorze OpenTelemetry. Źródło: [49]

Listing 3 Konfiguracja pipeline'ów OpenTelemetry Collector w prototypie. Wariant produkcyjny z użyciem stosu Grafana Cloud observability. Źródło: Opracowanie własne.

```
extensions:
  basicauth/client:
    client_auth:
      username: "${GRAFANA_INSTANCE_ID}"
      password: "${GRAFANA_TOKEN}"
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:9001
      http:
        endpoint: 0.0.0.0:9000
  prometheusremotewrite:
    endpoint: 0.0.0.0:9003

processors:
  batch:

exporters:
  otlphttp:
    endpoint: "https://${GRAFANA_ENDPOINT}"
    auth:
      authenticator: basicauth/client

service:
  extensions: [basicauth/client]
  pipelines:
    metrics:
      receivers: [otlp,prometheusremotewrite]
      processors: [batch]
      exporters: [otlphttp]
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [otlphttp]
    logs:
      receivers: [otlp]
      processors: [batch]
      exporters: [otlphttp]
```

W prototypie w środowisku lokalnym, kolektor rozdziela trzy rodzaje danych do trzech samodzielnie hostowanych systemów otwartoźródłowych:

- metryki kierowane są do bazy szeregów czasowych (TSDB) systemu Prometheus (podrozdział 4.19)
- ślady trafiają do systemu Jaeger (podrozdział 4.20)
- logi przekazywane są do systemu Grafana Loki (podrozdział 4.22)

W potoku produkcyjnym, prezentowanym na Listingu 3, wszystkie dane trafiają do jednej końcówki dostawcy chmurowego, co zanotowane jest jako zmienna `${GRAFANA_ENDPOINT}`, a następnie trasowane są do wewnętrznego systemu Prometheus, Grafana Loki oraz Grafana Tempo (chmurowy konkurent systemu Jaeger opisanego wyżej)

Zgodnie z przyjętą w niniejszym rozdziale doktryną kompromisu, również projekt warstwy obserwowalności wymaga uczciwego zestawienia korzyści z kosztami. Po stronie zalet należy

wymienić przede wszystkim oparcie o otwarty standard, eliminujące uzależnienie od dostawcy, jednolitą i bezobsługową dla zespołu instrumentację, kompletność obrazu systemu obejmującego wszystkie komponenty oraz centralizację wglądu w postaci jednego narzędzia wizualizacji. Rozwiązanie to bezpośrednio adresuje problem diagnostyczny będący jednym z fundamentalnych kosztów architektury mikroserwisowej.

6. Implementacja systemu

Niniejszy rozdział opisuje szczegóły implementacji systemu Glowing Waffle AMS w obszarach wzorców projektowych, warstwy backendowej, frontendowej, mechanizmów bezpieczeństwa, konteneryzacji oraz konfiguracji środowisk uruchomieniowych. Warstwa backendowa systemu składa się z sześciu mikroservisów napisanych w języku Java z wykorzystaniem frameworka Spring Boot, komunikujących się asynchronicznie za pośrednictwem brokera RabbitMQ oraz synchronicznie poprzez bramę API Traefik. Warstwa frontendowa zrealizowana jest jako aplikacja jednostronicowa (ang. *Single Page Application*, SPA) zbudowana w oparciu o framework Next.js z językiem TypeScript. Całość środowiska uruchomieniowego opiera się na platformie Docker zarządzanej przy pomocy Docker Compose. Architekturę systemu wraz z wyodrębnionymi komponentami i przepływami komunikacyjnymi przedstawiono na Rysunku 11 w rozdziale 5.

6.1. Wzorce projektowe użyte w systemie

Wzorce projektowe stanowią udokumentowane, powtarzalne rozwiązania typowych problemów pojawiających się podczas tworzenia projektów programistycznych. Ich znajomości i stosowanie porządkują strukturę kodu, ułatwiają jego zrozumienie, usprawniają pracę i ograniczają problemy we współpracy zespołowej, a także skracają czas wytwarzania dzięki sięganiu po rozwiązania o znanych właściwościach. Według autorów książki „Design Patterns: Elements of Reusable Object-Oriented Software” [43] – którzy jako pierwsi skatalogowali i ujednoliciли klasyczne wzorce projektowania programowania obiektowego – wyróżnia się trzy grupy wzorców:

- **kreacyjne** – wprowadzają elastyczne mechanizmy tworzenia obiektów i ponowne wykorzystywanie kodu,
- **strukturalne** – dotyczą sposobów komponowania klas i obiektów w większe struktury, z zachowaniem przejrzystości, elastyczności i możliwości dalszej rozbudowy
- **czynnościowe** – zajmują się efektywną komunikacją i współpracą między obiektową.

Implementacja systemu Glowing Waffle AMS wykorzystuje wzorce na dwóch poziomach: obiektowym oraz warstwy aplikacji. Wzorce o charakterze architektonicznym – takie jak Database per Service, Saga czy API Gateway – omówiono w rozdziale 5, niniejszy podrozdział koncentruje się na wzorcach poziomu kodu. Część z nich dostarczana jest bezpośrednio przez *framework* Spring, część zaś zaimplementowano samodzielnie – przy każdym zaznaczono jego rolę w systemie.

6.1.1 Wzorzec specyfikacji

Złożone wyszukiwanie obiektów według wielu opcjonalnych kryteriów często prowadzi do wzrostu logiki warunkowej zapytań. Wzorzec specyfikacji (ang. *Specification Pattern*) rozwiązuje ten problem, pozwalając wyrazić regułę wyboru obiektów jako samodzielny, komponowalny predykat – odseparowany zarówno od encji, jak i od logiki budowy zapytania. Wzorzec ten nie należy do klasycznego katalogu GoF, lecz wywodzi się z projektowania dziedzicznego (ang. *Domain-Driven Design*) [43] ze względu na sposób działania, jest mu najbliższy do wzorców o charakterze czynnościowym.

W systemie zastosowano go do dynamicznego wyszukiwania środków trwałych, udostępnionego przez końcówkę POST /assets/search. Każde kryterium filtra – zakres ceny, nazwa, stan techniczny, lokalizacja czy wartość pola niestandardowego – jest osobnym obiektem *Specification<Asset>*, a metoda *byFilter* scala je operatorem *and* w jedno zapytanie. Kryteria nieprzekazane zwracają wartość *null* i są pomijane, dzięki czemu z opcjonalnych pól filtra powstaje dokładnie takie zapytanie, jakie wynika z przesłanych parametrów. Fragment implementacji przedstawiono na Listingu 4.

Listing 4 Implementacja wzorca Specyfikacji. Źródło: Opracowanie własne.

```
public class AssetSpecification {
    public static Specification<Asset> byFilter(AssetSearchFilter filter) {
        if (filter == null) {
            return null;
        }

        return Specification.where(conditionsIn(filter.getAssetConditions()))
            .and(tagsIn(filter.getAssetTags()))
            .and(nameLike(filter.getName()))
            .and(assignedEquals(filter.getAssigned()))
            .and(responsibleUnitIn(filter.getResponsibleUnitUuids()))
            .and(unitPriceBetween(filter.getUnitPriceFrom(),
filter.getUnitPriceTo()))
            // pozostałe kryteria pominięto
        }

    private static Specification<Asset> assignedEquals(Boolean assigned) {
        return (root, query, criteriaBuilder) -> {
            if (assigned == null) {
                return null;
            }
            return criteriaBuilder.equal(root.get("assigned"), assigned);
        };
    }

    private static Specification<Asset> nameLike(String name) {
        return (root, query, criteriaBuilder) -> {
            if (name == null || name.isBlank()) {
                return null;
            }
            String pattern = "%" + name.toLowerCase() + "%";
            return criteriaBuilder.like(criteriaBuilder.lower(root.get("name")),
pattern);
        };
    }

    private static Specification<Asset> hasCustomData(Map<String, String> customData)
    {
        return (root, query, criteriaBuilder) -> {
            if (customData == null || customData.isEmpty()) {
                return null;
            }

            return customData.entrySet().stream().map(entry -> {
                var jsonExtract =
criteriaBuilder.function("jsonb_extract_path_text", String.class,
                    root.get("customData"),
                    criteriaBuilder.literal(entry.getKey()));

                return criteriaBuilder.equal(jsonExtract, entry.getValue());
            })
                .reduce(criteriaBuilder::and)
                .orElse(null);
        };
    }
}
```

```
// [...] pozostałe kryteria pominięto  
}
```

Tak skonstruowana specyfikacja jest następnie przekazywana do repozytorium rozszerzającego interfejs `JpaSpecificationExecutor<Asset>` – konkretnie do wbudowanej metody `Page<T> findAll(Specification<T> spec, Pageable pageable)`, realizującej także stronicowanie wyników. Warto odnotować, że jedno z kryteriów (`hasCustomData`) filtruje również po polach niestandardowych środka trwałego przechowywanych w kolumnie typu `jsonb`, wykorzystując funkcję `jsonb_extract_path_text`.

6.1.2 Wzorzec adaptera

Współpraca z systemami zewnętrznymi często wiąże się z koniecznością komunikacji niezgodny ze sobą interfejsów. Wzorzec adaptera (ang. *Adapter Pattern*) rozwiązuje ten problem, dopasowując interfejs, z którego pochodzą dane, do interfejsu oczekiwanego przez klienta. W klasyfikacji GoF [43] adapter zaliczany jest do wzorców strukturalnych, ponieważ dotyczy sposobu łączenia obiektów o niezgodnych interfejsach w jedną, współpracującą strukturę.

Tę rolę pełni w systemie klasa `KeycloakRoleConverter`, która implementuje interfejs `Converter<Jwt, Collection<GrantedAuthority>>` frameworka Spring i tłumaczy niestandardową strukturę `resource_access` tokenu Keycloak na ma model uprawnień `GrantedAuthority` rozumiany przez Spring Security. Bez tej adaptacji role osadzone w tokenie nie zostałyby rozpoznane przez mechanizmy autoryzacji. Listing 5 prezentuje implementację tego wzorca.

Listing 5 Klasa `KeycloakRoleConverter` realizująca wzorzec adaptera. Źródło: Opracowanie własne.

```
public class KeycloakRoleConverter implements Converter<Jwt,  
Collection<GrantedAuthority>> {  
  
    @Override  
    public Collection<GrantedAuthority> convert(Jwt jwt) {  
        Map<String, Object> resourceAccess =  
jwt.getClaimAsMap("resource_access");  
  
        if (resourceAccess == null || resourceAccess.isEmpty()) {  
            return Collections.emptyList();  
        }  
  
        Map<String, Object> roleMap = (Map<String, Object>)  
resourceAccess.get("nextjs-app");  
        Collection<String> roles = (List<String>) roleMap.get("roles");  
  
        if (roles == null || roles.isEmpty()) {  
            return Collections.emptyList();  
        }  
  
        return roles.stream()  
            .map(role -> new SimpleGrantedAuthority("ROLE_" + role))  
            .collect(Collectors.toList());  
    }  
}
```

6.1.3 Wzorzec fasady

W rozbudowanych *frameworkach* dostęp do wewnętrznych mechanizmów bywa złożony i rozproszony, a bezpośrednie korzystanie z niego przez warstwę domenową prowadzi do silnej zależności od infrastruktury. Wzorzec fasady (ang. *Facade Pattern*) udostępnia prosty, jednolity interfejs ukrywając pod spodem złożone podsystemy czy operacje na których klient nie powinien operować bezpośrednio. Zgodnie z klasyfikacją GoF [43] fasada należy do grupy wzorców strukturalnych, gdyż porządkuje sposób, w jaki klient odwołuje się do złożonej struktury podsystemów.

W systemie Glowing Waffle AMS reprezentuje go m.in. komponent `SecurityComponent`, ukrywając szczegóły dostępu do kontekstu bezpieczeństwa Spring Security (`SecurityContextHolder`) – w tym operacje takie jak dekodowanie tokenu JWT, pobieranie i walidowanie informacji pozyskanych z tokenu – ukrywając je za prostymi metodami takimi jak `getCurrentUserId` czy `getCurrentUserEmail`. Dzięki temu serwisy domenowe pobierają tożsamość zalogowanego użytkownika bez bezpośredniej zależności od wewnętrznych mechanizmów frameworka, a logika odczytu tożsamości pozostaje w jednym miejscu pozwalając się jedynie odwoływać do potrzebnych metod. Listing 6 prezentuje implementację tego wzorca.

Listing 6 Komponent `SecurityComponent` jako fasada kontekstu bezpieczeństwa. Źródło: Opracowanie własne.

```
@Component
public class SecurityComponent {

    public UUID getCurrentUserId() {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();

        if (authentication == null || !(authentication.getPrincipal() instanceof
Jwt jwt)) {
            throw new AccessDeniedException("No security context found - expected
JWT");
        }

        try {
            return UUID.fromString(jwt.getSubject());
        } catch (IllegalArgumentException e) {
            throw new AccessDeniedException("Claim 'sub' from JWT is not valid
UUID.", e);
        }
    }

    public String getCurrentUserEmail() {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();

        if (authentication == null || !(authentication.getPrincipal() instanceof
Jwt jwt)) {
            throw new AccessDeniedException("No security context found - expected
JWT");
        }

        return jwt.hasClaim("email") ? (String) jwt.getClaim("email") : null;
    }
    // pozostałe metody dostępne pominięt
}
```

6.1.4 Wzorzec Obserwator

Powiadamianie wielu elementów systemu o wystąpieniu zdarzenia bez tworzenia sztywnych zależności między nimi wymaga mechanizmu, który nada luźne powiązanie między nadawcą a odbiorcami. Wzorzec obserwatora (ang. *Observer Pattern*) definiuje zależność jeden-do-wielu, w której wywołanie ustalonej wcześniej czynności skutkuje powiadomieniem zarejestrowanych subskrybentów o aktywności. W katalogu GoF [43] obserwator zaliczany jest do wzorców czynnościowych, ponieważ definiuje sposób komunikacji i przepływu powiadomień między współpracującymi obiektami.

W warstwie backendowej wykorzystano wbudowany mechanizm zdarzeń frameworka Spring: serwis domenowy publikują zdarzenie wywołaniem `ApplicationEventPublisher.publishEvent(...)`, a dedykowany komponent reaguje na nie metodą oznaczoną adnotacją `@TransactionalEventListener`. Nasłuch skonfigurowane na fazę `AFTER_COMMIT`, dzięki czemu zdarzenie trafia do brokera RabbitMQ dopiero po pomyślnym zatwierdzeniu transakcji bazodanowej – co realizuje regułę publikacji zdarzeń opisaną w podrozdziale 5.5 i odgrywa logikę domenową od warstwy komunikacji asynchronicznej. Mechanizm przedstawiono na Listingu 7.

Listing 7 Realizacja wzorca obserwatora na przykładzie AssetHistoryPublisher. Źródło: Opracowanie własne.

```
@Slf4j
@Component
@RequiredArgsConstructor
public class AssetHistoryPublisher {

    private static final String EVENT_EXCHANGE = "domain.events";
    private static final String ROUTING_KEY = "event.asset.history";

    private final RabbitTemplate rabbitTemplate;

    @Async
    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    public void handleAssetHistoryEvent(AssetHistoryEvent event) {
        log.info("DB transaction accepted. Sending AssetHistoryEvent [action={},
assetId={}] to RabbitMQ...",
            event.action(), event.assetId());

        rabbitTemplate.convertAndSend(EVENT_EXCHANGE, ROUTING_KEY, event);
    }
}
```

6.2. Implementacja infrastruktury serwerowej – backend

Warstwa backendowa systemu Glowing Waffle AMS zrealizowana została jako zbiór niezależnych mikroservisów napisanych w języku Java w wersji 25 z wykorzystaniem frameworka Spring Boot. Każdy z mikroservisów stanowi samodzielną aplikację o ściśle wyznaczonych granicach kontekstu dziedziny, posiadającą własną relacyjną bazę danych PostgreSQL oraz komunikującą się z pozostałymi serwisami wyłącznie poprzez broker wiadomości RabbitMQ. Wewnętrzna budowa pojedynczego mikroservisów opiera się na architekturze MVC z wyraźnym rozdziałem odpowiedzialności pomiędzy warstwę udostępniania interfejsu REST, warstwę logiki dziedziny oraz warstwę trwałości. Niniejszy podrozdział przedstawia kolejno strukturę projektu i podział na moduły,

warstwową budowę mikroservisu, model dziedziny wraz z warstwą trwałości, mechanizmy walidacji i obsługi błędów, a także implementację komunikacji asynchronicznej oraz synchronicznych zapytań w stylu RPC. Decyzje architektoniczne stojące u podstaw tych rozwiązań omówiono w rozdziale 5, a zastosowane wzorce projektowe poziomu kodu – w podrozdziale 6.1, natomiast mechanizmy bezpieczeństwa – w podrozdziale 6.4; niniejszy podrozdział koncentruje się na ich realizacji w kodzie.

6.2.1 Struktura projektu

Struktura została zorganizowana jako wielomodułowy projekt Maven 4 (ang. *multimodule project*). Nadrzędny moduł (ang. *root*) o identyfikatorze `com.pjatk:ams` i typie pakowania POM (ang. *Project Object Model*) nie zawiera kodu, lecz pełni rolę spajającą dla pozostałych modułów Mavenowych, oraz zapewnia dziedziczenie modułu Spring Boot w wersji 4.0.5. Dzięki temu, wersja języka Java (25) oraz zależności są centralnie obsługiwane w jednym miejscu. Istotną część centralnego pliku `pom.xml` przedstawiono na Listingu 8.

Listing 8 Plik POM (Project Object Model) nadrzędnego projektu prototypu. Źródło: Opracowanie własne.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>4.0.5</version>
  <relativePath/>
</parent>
<modules>
  <module>asset-service</module>
  <module>inventory-service</module>
  <module>loan-service</module>
  <module>notification-service</module>
  <module>reporting-service</module>
  <module>incident-service</module>
  <module>kc-proxy</module>
  <module>ams-shared-api</module>
</modules>
<groupId>com.pjatk</groupId>
<artifactId>ams</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>Asset Management System</name>
<description>Engineering thesis</description>
<url/>
<scm>
  <connection>scm:git:https://github.com/Patkaniec22/glowing-
waffle.git</connection>
  <developerConnection>scm:git:ssh://git@github.com/Patkaniec22/glowing-
waffle.git</developerConnection>
  <url>https://github.com/Patkaniec22/glowing-waffle</url>
  <tag>HEAD</tag>
</scm>

<properties>
  <java.version>25</java.version>
</properties>
<packaging>pom</packaging>
```

Na strukturę projektu składa się sześć mikroservisów biznesowych, pomocniczy serwis `kc-proxy` oraz biblioteka współdzielonych kontraktów `ams-shared-api`. Liczba mikroservisów wynika bezpośrednio z analizy tzw. granicy kontekstu, co już omówiono w rozdziale 5.1. Aplikacja kliencka w technologii Next.js stanowi odrębny projekt i nie jest częścią ekosystemu Maven.

Strukturę pojedynczego mikroservisów zaprojektowano z podziałem na odpowiedzialności danych klas; zaprezentowano to, wraz z krótkim opisem każdego pakietu, na Listingu 9.

Listing 9 Struktura pakietów mikroservisów Asset Service. Źródło: Opracowanie własne.

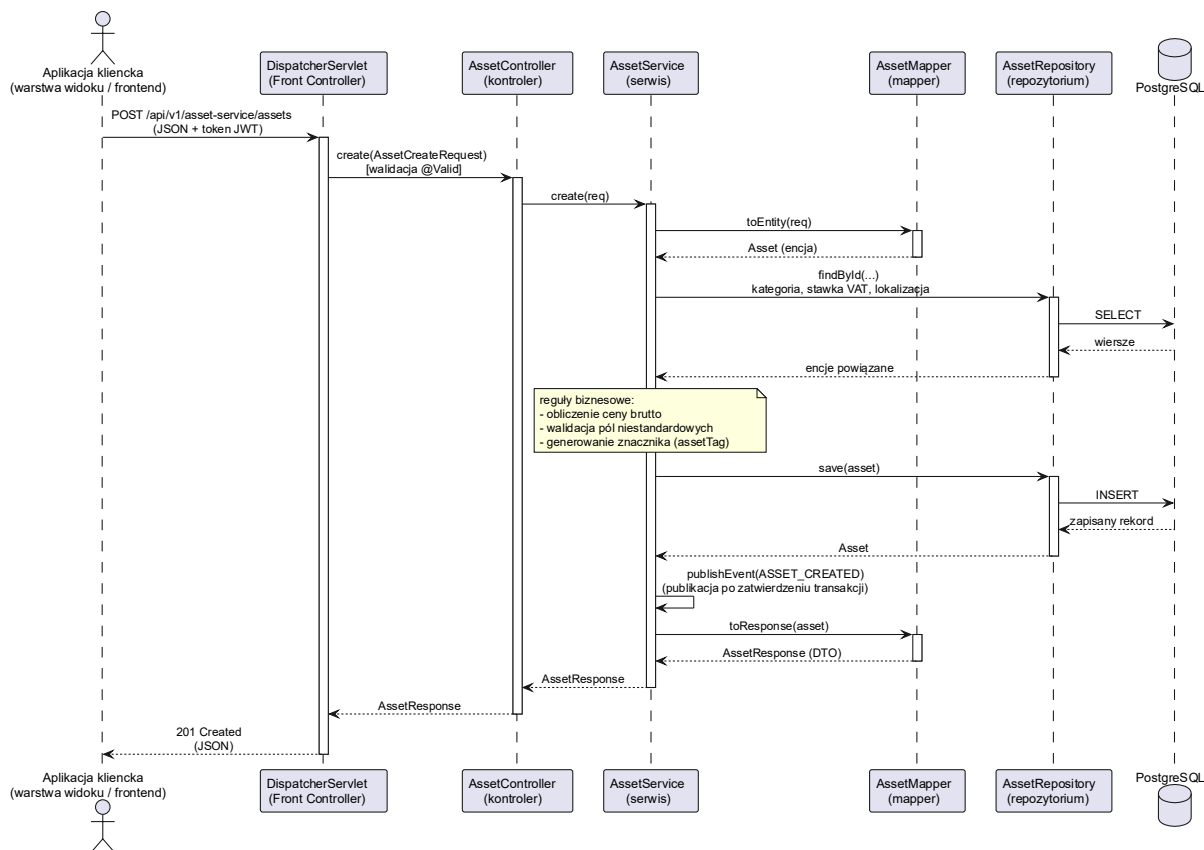
```
com.pjatk.ams.assetservice
├── config           // konfiguracja (Rabbit, Security, inicjalizacja danych)
├── controller       // kontrolery REST oraz kontroler RPC
├── service          // logika dziedzinowa
├── repository       // repozytoria Spring Data JPA
├── model            // encje JPA oraz typy enum
├── dto              // obiekty transferu danych
├── mapper           // mappery biblioteki MapStruct (encja <-> DTO)
├── specification    // wzorzec specyfikacji
├── validation       // własne adnotacje i walidatory
├── advice           // globalna obsługa wyjątków
├── rabbit           // klasy produkujące zdarzenia do brokera
├── listener         // klasy konsumentów zdarzeń z brokera
├── converter        // konwerter ról tokenów JWT
└── ┬── utils        // komponenty pomocnicze (np. SecurityComponent)
```

Przyjęty układ pakietów jest powtarzany w pozostałych mikroservisach, co zapewnia jednolitą konwencję i ułatwia nawigację po kodzie niezależnie od obsługiwanej dziedziny. Różnice sprowadzają się do pakietów specyficznych dla danej funkcjonalności (np. `scheduler` w `loan-service`, `integration` w `inventory-service` czy konfiguracja składowania plików w `incident-service`, omówione w podrozdziale 6.2.7). Każdy mikroservis deklaruje zależność od biblioteki `ams-shared-api`, dzięki czemu współdzielone kontrakty komunikacyjne pozostają jednolite dla całego systemu.

6.2.2 Architektura MVC

Warstwę serwerową systemu Glowing Waffle AMS zbudowano na frameworku Spring MVC, realizującym obsługę żądań HTTP zgodnie z wzorcem Model-Widok-Kontroler. W systemie wzorzec ten przyjmuje postać rozproszoną: warstwę widoku (ang. *View*) stanowi w całości aplikacja kliencka opisana w podrozdziale 6.3, natomiast backend dostarcza warstwę kontrolera (ang. *Controller*) oraz modelu (ang. *Model*). Obie strony w zakresie danych domenowych komunikują się przez interfejs REST z wymianą danych w formacie JSON – serwer nie generuje żadnych widoków. Punktem wejścia po stronie serwera jest centralny kontroler `DispatcherServlet`, który kieruje każde żądanie do właściwego kontrolera, a zwróconą wartość serializuje do formatu JSON przy użyciu konwerterów komunikatów (`HttpMessageConverter`).

Obsługa pojedynczego żądania przechodzi przez stałą, powtarzalną w każdej usłudze sekwencję warstw: kontroler → serwis → repozytorium → encja → mapper → DTO. W kolejnych punktach omówiono rolę każdej z nich, śledząc na bieżąco przykład utworzenia środka trwałego (POST `/api/v1/asset-service/assets`) oraz jego odczytu (GET `.../assets/{uuid}`). Przepływ ten zilustrowano na Rysunek 37.



Rysunek 37 Przeływ żądania utworzenia środka trwałego (POST /assets). Źródło: opracowanie własne

Warstwa kontrolera

Kontroler stanowi cienką warstwę wejścia, oznaczoną adnotacją `@RestController` i wspólnym prefiksem ścieżki w konwencji `/api/v1/{usługa}/{zasób}`, co zapewnia spójne, wersjonowane API. Nie zawiera logiki biznesowej - jego rolą jest przyjęcie żądania, delegacja do warstwy serwisowej i zwrócenie obiektu transferu danych (DTO). Metody HTTP odwzorowano na operacje na zasobie (GET, POST, PUT, DELETE), a kody odpowiedzi ustawiono jawnie adnotacją `@ResponseStatus` (np. 201 Created). Ciało żądania wiązane jest adnotacją `@RequestBody` i poddawane walidacji przez `@Valid` (podrozdział 0), a zależności wstrzykiwane są przez konstruktor. Przykład kontrolera pokazano na Listingu 10.

Listing 10 Skrócony fragment kontrolera `AssetController` – odwzorowanie metod HTTP na operacje CRUD. Źródło: Opracowanie własne.

```

@RestController
@RequestMapping("/api/v1/asset-service/assets")
@RequiredArgsConstructor
public class AssetController {
    private final AssetService assetService;

    @GetMapping("/{uuid}")
    public AssetResponse getAssetById(@PathVariable UUID uuid) {
        return assetService.findById(uuid);
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public AssetResponse createAsset(@Valid @RequestBody AssetCreateRequest req) {
        return assetService.create(req);
    }
}

```

```

public AssetResponse create(@Valid @RequestBody AssetCreateRequest req) {
    return assetService.create(req);
}

@PutMapping("/{uuid}")
@ResponseStatus(HttpStatus.OK)
public AssetResponse update(@PathVariable UUID uuid,
                           @Valid @RequestBody AssetUpdateRequest req) {
    return assetService.update(uuid, req);
}

@DeleteMapping("/{uuid}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void delete(@PathVariable UUID uuid) {
    assetService.delete(uuid);
}
}

```

Warstwa serwisowa

Warstwa serwisowa (@Service) skupia logikę biznesową i wyznacza granice transakcji (@Transactional). W przykładzie tworzenia środka serwis odwzorowuje żądanie na encję, pobiera powiązane encje (kategorię, stawkę VAT, lokalizację) z repozytoriów, przetwarza operacje dziedzinowe – m.in. wylicza cenę brutto na podstawie stawki VAT oraz weryfikuje, czy przekazane pola niestandardowe są dozwolone dla danej kategorii bądź typu - generuje znacznik (assetTag), utrzuca encję, publikuje zdarzenie domenowe (podrozdział 6.2.5), po czym zwraca odpowiedź w postaci DTO. Listing 11 przedstawia okrojony w celu uproszczenia metodę create() dobrze odzwierciedlającą przepływ tej operacji.

Listing 11 Skrócony fragment serwisu AssetService – tworzenie środka trwałego. Źródło: opracowanie własne

```

@Service
@RequiredArgsConstructor
public class AssetService {
    private final AssetRepository assetRepo;
    private final AssetCategoryRepository assetCategoryRepo;
    private final VatRateRepository vatRateRepo;
    private final AssetMapper mapper;
    private final AssetTagGenerator assetTagGenerator;

    @Transactional
    public AssetResponse create(AssetCreateRequest req) {
        Asset entity = mapper.toEntity(req); // DTO -> encja

        AssetCategory cat = assetCategoryRepo.findById(req.getCategoryUuid())
            .orElseThrow(
                () -> new IllegalArgumentException("AssetCategory not found"));
        entity.setAssetCategory(cat);
        // [...]
        VatRate vatRate = vatRateRepo.findById(req.getVatRateUuid())
            .orElseThrow(
                () -> new IllegalArgumentException("VatRate not found"));
        entity.setVatRate(vatRate);
    }
}

```

```

        entity.setUnitPriceGross(req.getUnitPriceNet()           // reguła
biznesowa
        .multiply(BigDecimal.ONE.add(vatRate.getRate()))
        .setScale(2, RoundingMode.HALF_UP));

        entity.setAssetTag(assetTagGenerator.generateTag(entity));
        Asset saved = assetRepo.save(entity);                    // utrwalenie

        publishHistoryEvent(saved, "ASSET_CREATED", buildSnapshot(saved));
        return mapper.toResponse(saved);                          // encja -> DTO
    }
}

```

Dla operacji odczytu przepływ jest analogiczny, lecz znacznie krótszy – serwis pobiera encję z repozytorium i odwzorowuje ją na DTO, a w przypadku braku rekordu zgłasza wyjątek tłumaczony następnie na odpowiedź HTTP (podrozdział 0), co zostało przedstawione na Listingu 12.

Listing 12 Skrócony fragment serwisu `AssetService` - odczyt środka trwałego i odwzorowanie encji na DTO. Źródło: Opracowanie własne.

```

@Transactional(readOnly = true)
public AssetResponse findById(UUID id) {
    log.info("Finding asset by id {}", id);
    return assetRepo.findById(id)
        .map(mapper::toResponse)
        .orElseThrow(
            () -> new IllegalArgumentException("Asset not found: " + id));
}

```

Warstwa repozytorium

Dostęp do bazy danych zrealizowano przez repozytoria Spring Data JPA – interfejsy rozszerzające `JpaRepository` (podstawowe operacje CRUD) oraz `JpaSpecificationExecutor`, umożliwiające wykonywanie dynamicznych zapytań budowanych wzorcem specyfikacji (podrozdział 6.1.1). Przyjęto przy tym zasadę wyboru najprostszego narzędzia adekwatnego do problemu. Najprostsze zapytania definiuje się jako metody pochodne z nazwy (np. `existsByAssetTag`), z których Spring Data samodzielnie wyprowadza treść - nie wymagają one zapytania w postaci łańcucha znaków i są weryfikowane już przy starcie aplikacji – w istocie znaczna część repozytoriów systemu nie zawiera ani jednego jawnego zapytania. Bardziej złożone przypadki zapisuje się jawnie w języku JPQL (`@Query`) - m.in. pobranie z dociągnięciem relacji (`LEFT JOIN FETCH`) ograniczające problem "n+1", czy wyszukiwanie wzorcem (`LIKE`).

Operacje modyfikujące wiele rekordów jednocześnie wymagają dodatkowo adnotacji `@Modifying`. Domyślnie metoda opatrzona `@Query` traktowana jest jako zapytanie wybierające (`SELECT`) zwracające encje – `@Modifying` informuje Spring Data, że zapytanie zmienia stan bazy (`UPDATE/DELETE`), a metoda zwraca liczbę zmodyfikowanych wierszy (bądź `void`). Pozwala to przeprowadzić masową aktualizację jednym poleceniem SQL – bez wczytywania każdej encji do kontekstu trwałości – co w systemie wykorzystano np. do oznaczania środków objętych trwającą kampanią inwentaryzacyjną. W systemie metody te wywoływane są w obrębie transakcji w reakcji na zdarzenia kampanii inwentaryzacyjnej (blokowanie i odblokowywanie środków na czas inwentaryzacji), a zwracana liczba zmodyfikowanych wierszy służy do potwierdzenia, że operacja objęła wszystkie wskazane środki. Listing 13 przedstawia wybrane metody tej implementacji.

Listing 13 Wybrane metody repozytorium AssetRepository. Źródło: Opracowanie własne.

```
public interface AssetRepository extends JpaRepository<Asset, UUID>,
JpaSpecificationExecutor<Asset> {

    boolean existsByAssetTag(String assetTag);           // zapytanie pochodne z
nazwy

    @Query("SELECT a FROM Asset a " +
            "LEFT JOIN FETCH a.purchase " +
            "LEFT JOIN FETCH a.location " +
            "LEFT JOIN FETCH a.assetType " +
            "LEFT JOIN FETCH a.assetCategory " +
            "LEFT JOIN FETCH a.vatRate ")
    List<Asset> findAllWithEagerRelationships();          // JPQL + JOIN FETCH

    @Query("SELECT a FROM Asset a WHERE a.assetTag like :assetTag")
    Optional<Asset> findByAssetTag(String assetTag);

    @Modifying
    @Query(value = ""
            UPDATE Asset a
            SET a.duringInventory = TRUE
            WHERE a.id IN :ids
            """)
    int markAssetsAsDuringInventory(@Param("ids") Set<UUID> ids);
}
```

Wybór JPQL, a nie natywnego języka SQL, podyktowany był dwiema zaletami. Po pierwsze, JPQL operuje na modelu obiektowym – encjach i ich polach, a nie na tabelach i kolumnach - dzięki czemu pozostaje niezależny od dialektu konkretnej bazy danych i sprzyja przenośności rozwiązania. Po drugie, zapytania JPQL są walidowane przez dostawcę trwałości podczas uruchamiania aplikacji – błąd w nazwie pola czy encji ujawnia się natychmiast przy starcie, a nie dopiero w trakcie obsługi żądania. Natywny SQL zarezerwowano wyłącznie dla operacji wymagających funkcji specyficznych dla bazy PostgreSQL, których JPQL nie obejmuje – w całym systemie jest to pojedynczy przypadek operacji na kolumnie typu `jsonb`.

Świadomie zrezygnowano natomiast z budowania wszystkich zapytań w pełni typowanym interfejsem Criteria API. Choć – zwłaszcza w połączeniu z metamodeliem – eliminuje on ryzyko błędu w łańcuchu znaków na etapie kompilacji, jego programowy charakter czyni proste zapytania statyczne znacznie mniej czytelnymi i bardziej rozwlekłymi. Z tego powodu Criteria API ograniczono do zapytań rzeczywiście dynamicznych, konstruowanych w czasie wykonania z opcjonalnych kryteriów, co zrealizowano wzorcem specyfikacji opisanym w podrozdziale 6.1.1. Zastosowanie tego mechanizmu do zapytań statycznych stanowiłoby przykład nadmiernej inżynierii (ang. *overengineering*).

Należy przy tym uczciwie odnotować ograniczenie zapytań JPQL zapisanych jako łańcuch znaków: są one weryfikowane dopiero przez dostawcę trwałości (przy starcie aplikacji), a nie przez kompilator. W przypadku zapytań pobierających powiązane encje – takich jak pobranie środka trwałego wraz z jego relacjami – równoważnym, bardziej deklaratywnym narzędziem jest mechanizm grafów encji (ang. *Entity Graph*), pozwalający określić plan pobierania bez jawnego zapisywania klauzul JOIN FETCH. W projekcie pozostano przy zapisie JOIN FETCH, który utrzymuje plan pobierania bezpośrednio w treści zapytania, przejście na grafy encji stanowi naturalny kierunek dalszego uporządkowania tej warstwy.

Warstwa encji (model)

Trwały model dziedzinowy reprezentują encje JPA – obiekty Javy opatrzone adnotacjami mapowania obiektowo-relacyjnego, na które Hibernate (podrozdział 4.3) odwzorowuje wiersze tabel. Poniżej przytoczono jedynie szkielet encji środka trwałego, pełne omówienie modelu - relacji, kluczy, pól niestandardowych w jsonb – zawarto w podrozdziale 6.2.3. Listing 14 przedstawia implementację uproszczonego modelu dla encji Asset.

Listing 14 Uproszczony fragment encji Asset: wybrane pola i mapowanie obiektowo-relacyjne (JPA).
Źródło: Opracowanie własne.

```
@Entity
@Table(name = "asset")
@NoArgsConstructor @AllArgsConstructor
@Getter @Setter @Builder
public class Asset {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    @Column(name = "id", nullable = false)
    private UUID id;

    @Size(message = "Asset tag must be at least 2 characters long", min = 2)
    @Column(name = "asset_tag", nullable = false, unique = true)
    private String assetTag;

    @NotBlank(message = "Asset name cannot be empty")
    @Column(name = "name", nullable = false)
    private String name;

    @Enumerated(EnumType.STRING)
    @Column(name = "asset_condition", nullable = false)
    private AssetCondition assetCondition;

    @JdbcTypeCode(SqlTypes.JSON)
    @Column(name = "custom_data", columnDefinition = "jsonb")
    private Map<String, Object> customData = new HashMap<>();
    // pozostałe pola i relacje - por. 6.2.3
}
```

Warstwa mapowania

Granicę między modelem a światem zewnętrznym stanowią obiekty transferu danych, a ich tłumaczeniem w obie strony zajmuje się biblioteka MapStruct. Interfejs oznaczony adnotacją @Mapper(componentModel = "spring") jest implementowany automatycznie na etapie kompilacji, co eliminuje powtarzalny kod mapujący i zapewnia kontrolę typów. Zdefiniowano trzy podstawowe odwzorowania: utworzenie encji z żądania (toEntity), częściową aktualizację istniejącej encji z pominięciem pól pustych (updateEntity) oraz złożenie odpowiedzi z encji i jej relacji (toResponse). Fragment implementacji interfejsu został zaprezentowany na Listingu 15.

Listing 15 Skrócony fragment interfejsu mappera AssetMapper (MapStruct) z trzema podstawowymi odwzorowaniami. Źródło: Opracowanie własne.

```
@Mapper(componentModel = "spring")
public interface AssetMapper {
```

```

    @Mapping(target = "location", ignore = true)
    // [...]
    @Mapping(target = "unitPriceGross", ignore = true)
    @Mapping(target = "customData", source = "customData")
    @Mapping(target = "responsibleUnitId", source = "responsibleUnitUuid")
    Asset toEntity(AssetCreateRequest req);

    @BeanMapping(nullValuePropertyMappingStrategy =
    NullValuePropertyMappingStrategy.IGNORE)
    @Mapping(target = "location", ignore = true)
    // [...]
    @Mapping(target = "unitPriceGross", ignore = true)
    @Mapping(target = "customData", source = "customData")
    @Mapping(target = "responsibleUnitId", source = "responsibleUnitUuid")
    Asset updateEntity(@MappingTarget Asset asset, AssetUpdateRequest req);

    @Mapping(target = "locationUuid", source = "location.id")
    // [...]
    @Mapping(target = "vatRateUuid", source = "vatRate.id")
    @Mapping(target = "vatRateName", source = "vatRate.name")
    @Mapping(target = "currency", source = "purchase.currency")
    @Mapping(target = "customData", source = "customData")
    @Mapping(target = "responsibleUnitUuid", source = "responsibleUnitId")
    AssetResponse toResponse(Asset asset);
}

```

Obiekty transferu danych (DTO)

Obiekty transferu danych definiują kontrakt API niezależnie od schematu bazy, dzięki czemu encje nigdy nie są wystawiane bezpośrednio na zewnątrz. Rozróżniono DTO żądań (wejście, opatrzone regułami walidacji omówionymi w podrozdziale 0) oraz DTO odpowiedzi (wyjście). DTO odpowiedzi ma postać płaską – relacje reprezentowane są przez identyfikatory, a nie zagnieżdżone obiekty encji, co upraszcza kontrakt API, ogranicza ilość przesyłanych danych oraz zmniejsza ryzyko niekontrolowanego ujawnienia szczegółów modelu domenowego. Fragment implementacji obiektu żądania został zaprezentowany na Listingu 16.

Listing 16 Fragment obiektu żądania `AssetCreateRequest` z regułami walidacji. Źródło: Opracowanie własne.

```

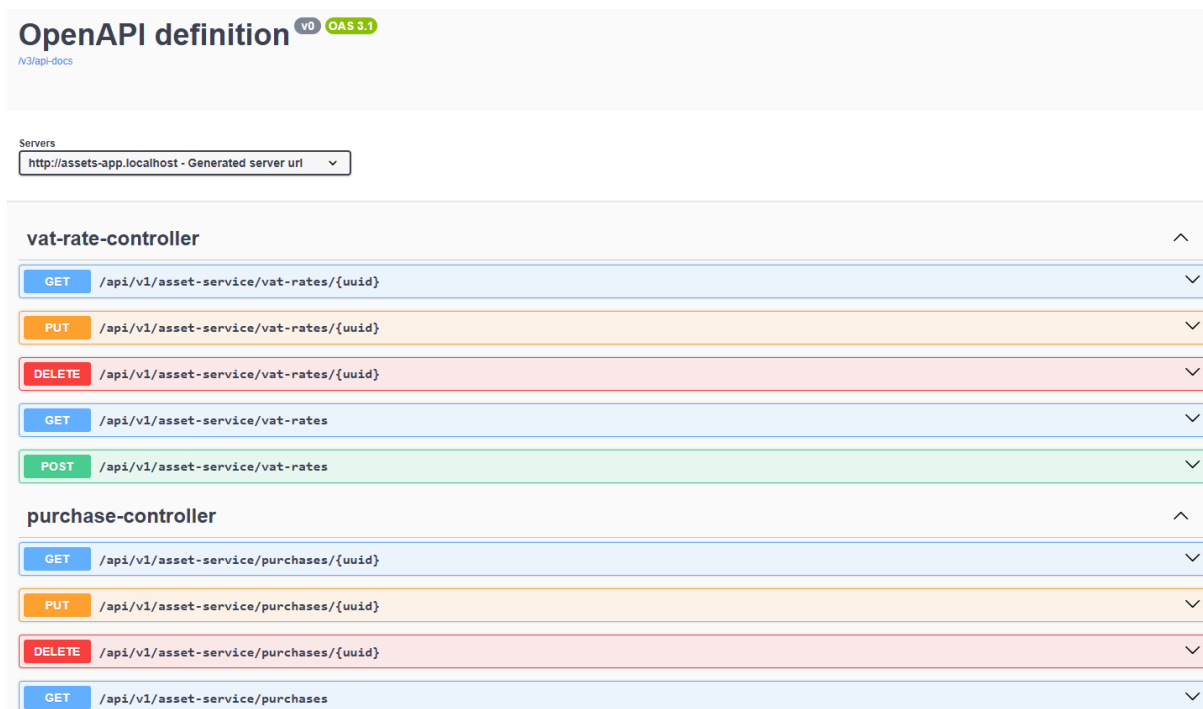
@Getter @Setter @Builder
@NoArgsConstructor @AllArgsConstructor
public class AssetCreateRequest {
    @NotBlank private String name;
    @NotNull private AssetCondition assetCondition;
    @Size(max = 255) private String description;
    @Positive private BigDecimal unitPriceNet;
    @NotNull private UUID vatRateUuid;
    @PastOrPresent(message = "Data nie może być z przyszłości")
    private LocalDateTime lastCampaignDate;
    private UUID responsibleUnitUuid;
    @NotNull private UUID categoryUuid;
    @NotNull private UUID purchaseUuid;
    @NotNull private UUID locationUuid;
    @NotNull private Boolean assigned;
    private UUID typeUuid;
    private Map<String, Object> customData;
}

```

```
}
```

Automatyczna dokumentacja REST

Dokumentację API wygenerowano automatycznie przy pomocy biblioteki `springdoc-openapi`. Na podstawie sygnatur kontrolerów oraz definicji typów DTO biblioteka tworzy specyfikację w standardzie OpenAPI 3 (ścieżka `/v3/api-docs`) oraz interaktywny interfejs Swagger UI (`/swagger-ui`), opisany jako narzędzie w podrozdziale 4.16. W projekcie nie zastosowano ręcznych adnotacji opisowych – dokumentacja powstaje wyłącznie na podstawie kodu, dzięki czemu pozostaje z nim trwale zgodna. Widok interfejsu Swagger UI zostało zaprezentowany na Rysunku 38.



Rysunek 38 Widok Swagger UI. Źródło: opracowanie własne

6.2.3 Encje i trwałość

Trwałość w projekcie zapewniono przy pomocy biblioteki Spring Data JPA. Encje reprezentowane są adnotacją `@Entity`, i mapowane przez framework Hibernate, który jest mapperem typu ORM. Schemat bazy danych generowany jest automatycznie na podstawie definicji klasy encji, co w warunkach prototypu upraszcza rozwój modelu. Kluczem głównym każdej encji został identyfikator typu UUID w wersji 4. Rezygnacja z centralnie generowanych, przyrostowych kluczy głównych postaci liczby całkowitej jest naturalną decyzją w architekturze mikroserwisowej, gdyż identyfikator można wygenerować niezależnie, lokalnie i bez koordynacji z bazą danych. Centralną encją mikroserwisu Asset Service jest środek trwały (Asset), powiązany asocjacjami z innymi encjami dziedziny. Skróconą definicję tej encji, wraz z przykładami atrybutów wymagalnych oraz asocjacjami zaprezentowano na Listingu 17, a graficznie przedstawiono na Rysunek 39.

Listing 17 Skrócony fragment encji Asset z asocjacjami oraz atrybutami niestandardowymi zapisywanymi jako JSONB. Źródło: Opracowanie własne.

```
@Entity
@Table(name = "asset")
```

```

@Getter @Setter @Builder
@NoArgsConstructor @AllArgsConstructor
public class Asset {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @ManyToOne(optional = false)
    @JoinColumn(name = "asset_category_id", nullable = false)
    private AssetCategory assetCategory; // asocjacja wymagana, jeden do wiele

    @ManyToOne(optional = true)
    private AssetType assetType; // asocjacja opcjonalna, jeden
do wiele

    @ManyToOne(optional = false)
    @JoinColumn(name = "vat_rate_id", nullable = false)
    private VatRate vatRate;

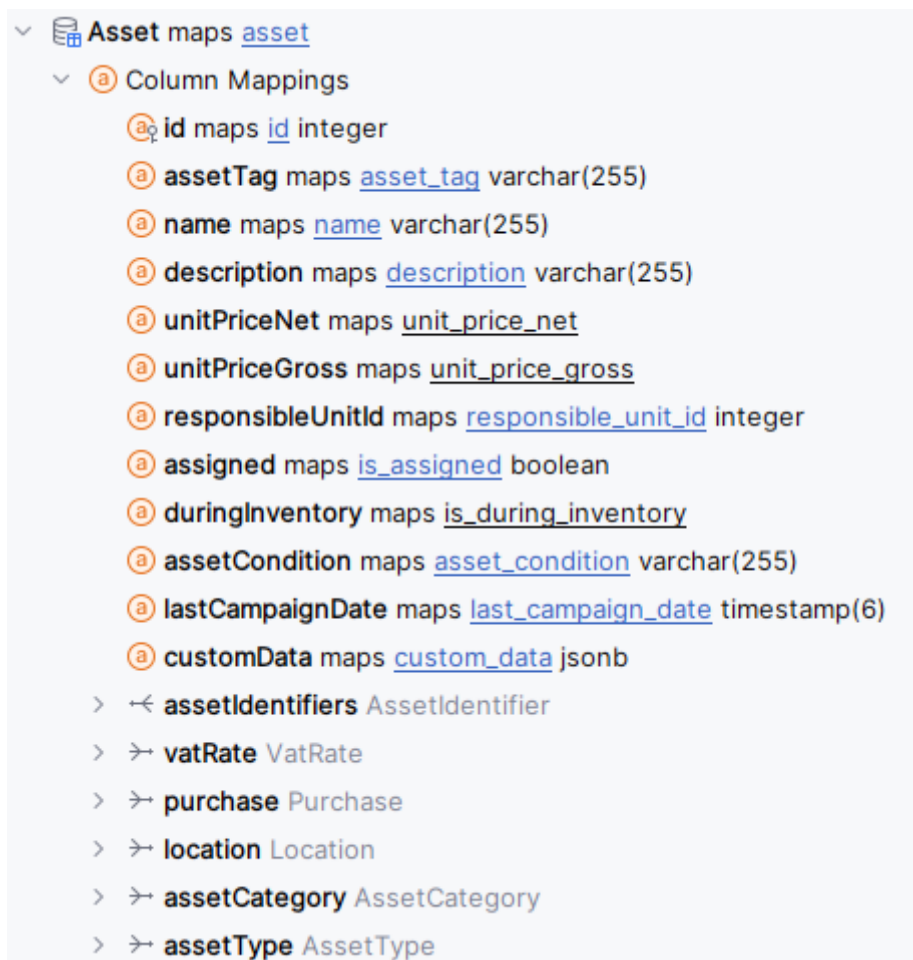
    @Column(name = "is_during_inventory", nullable = false)
    private Boolean duringInventory = false;
        // atrybut wymagalny

    @ManyToOne(optional = false) private Purchase purchase; // asocjacja
wymagana, jeden do wiele
    @ManyToOne(optional = false) private Location location; // asocjacja
wymagana, jeden do wiele

    @OneToMany(mappedBy = "asset", orphanRemoval = true)
    private List<AssetIdentifier> assetIdentifiers = new ArrayList<>(); //
asocjacja opcjonalna wiele do jeden

    @JdbcTypeCode(SqlTypes.JSON)
    @Column(name = "custom_data", columnDefinition = "jsonb")
    private Map<String, Object> customData = new HashMap<>();
    // pozostałe pola pominięto [...]
}

```



Rysunek 39 Encja Asset wraz z asocjacjami i mapowaniami typu ORM. Źródło: Opracowanie własne.

Atrybuty niestandardowe w kolumnie JSONB

Kluczowym mechanizmem realizującym wymóg elastyczności modelowania dziedziny jest przechowywanie tychże danych w pojedynczej kolumnie typu JSONB bazy PostgreSQL. Mapę customData odwzorowano na tę kolumnę adnotacją `@JdbcTypeCode(sqlTypes.JSON)`. Dodanie nowego atrybutu niestandardowego, nazwanego w projekcie polami niestandardowymi (ang. custom fields), nie wymaga zmiany schematu ani żadnej migracji tabeli. Istotnym jest również fakt, że możliwe jest filtrowanie po tej kolumnie – co opisane zostało w podrozdziale 6.1.1 przy okazji wzorca specyfikacji. Odbyna się to przy pomocy funkcji `jsonb_extract_path_text`. Metodę odpowiedzialną za wyszukiwanie w atrybutach niestandardowych zaprezentowano na Listingu 18.

Listing 18 Metoda przedstawiająca działanie wzorca Specyfikacji i wyszukiwania po atrybutach niestandardowych przy pomocy specjalnej funkcji PostgreSQL. Źródło: Opracowanie własne.

```

private static Specification<Asset> hasCustomData(Map<String, String>
customData) {
    return (root, query, criteriaBuilder) -> {
        if (customData == null || customData.isEmpty()) {
            return null;
        }

        return customData.entrySet().stream().map(entry -> {
  
```

```

        var jsonExtract =
criteriaBuilder.function("jsonb_extract_path_text", String.class,
                        root.get("customData"),
                        criteriaBuilder.literal(entry.getKey()));

        return criteriaBuilder.equal(jsonExtract, entry.getValue());
    })
        .reduce(criteriaBuilder::and)
        .orElse(null);

};
}

```

6.2.4 Weryfikacja danych przychodzących oraz obsługa błędów

Dane napływające do mikroservisów weryfikowane są dwuetapowo: deklaratywnie, na granicy warstwy kontrolera (walidacja obiektów żądania), oraz centralnie – przez globalny komponent przechwytyjący wyjątki i tłumaczący je na ujednolicone odpowiedzi HTTP. Takie rozdzielenie odciąża kontrolery i serwisy od powtarzalnej obsługi błędów oraz zapewnia spójny format komunikatów dla aplikacji klienckiej.

Walidacja na poziomie DTO

Reguły poprawności danych wejściowych wyrażono deklaratywnie, w postaci adnotacji standardu Jakarta Bean Validation, umieszczonych bezpośrednio na polach obiektów żądania (DTO). Adnotacja `@Valid` przy parametrze `@RequestBody()` uruchamia walidację jeszcze przed wejściem do ciała metody kontrolera. W obiekcie `AssetCreateRequest` (Listing 8) zastosowano m.in. `@NotBlank` (pole wymagane i niepuste), `@NotNull`, `@Size` (ograniczenie długości), `@Positive` (wartość dodatnia) oraz `@PastOrPresent` (data nie z przyszłości). Naruszenie którejkolwiek reguły przerywa przetwarzanie żądania i powoduje zgłoszenie wyjątku `MethodArgumentNotValidException`, obsługiwanego centralnie (opisanego w dalszej części podrozdziału).

Dedykowane walidatory

Gdy wbudowane adnotacje nie wystarczają, definiuje się własne ograniczenia (ang. *custom constraints*). Każde składa się z adnotacji opatrzonej `@Constraint(validatedBy = ...)` oraz klasy walidatora implementującej interfejs `ConstraintValidator`. W systemie zastosowano dwa rodzaje takich reguł.

Pierwszym jest walidacja pojedynczego pola – adnotacja `@UpperCase`, sprawdzająca, że łańcuch znaków zapisany jest wielkimi literami, co zostało zaprezentowane na Listingu 19.

Listing 19 Walidator pojedynczego pola: adnotacja `@UpperCase` wraz z walidatorem. Źródło: Opracowanie własne.

```

@Documented
@Constraint(validatedBy = UpperCaseValidator.class)
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface UpperCase {
    String message() default "Field must be in upper case";
}

```

```

    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}

public class UpperCaseValidator implements ConstraintValidator<UpperCase, String>
{
    @Override
    public boolean isValid(String value,
                           jakarta.validation.ConstraintValidatorContext context) {
        if (value == null || value.isBlank()) {
            return true;
        }
        return value.equals(value.toUpperCase());
    }
}

```

Drugim, bardziej rozbudowanym, jest walidacja międzypolowa nakładana na cały obiekt. Adnotacja `@GrossNotLessThanNet` o zasięgu `ElementType.TYPE` (nakładana na klasę) egzekwuje regułę biznesową, zgodnie z którą kwota brutto nie może być niższa od kwoty netto. Aby walidator pozostał niezależny od konkretnego typu DTO, porównywane pola udostępniono przez interfejs `NetGrossHolder`, który implementuje obiekty objęte tą regułą. Implementacja walidacji została przedstawiona na Listingu 20.

Listing 20 Walidacja międzypolowa: adnotacja klasowa `@GrossNotLessThanNet`, interfejs `NetGrossHolder` oraz walidator. Źródło: Opracowanie własne.

```

@Documented
@Constraint(validatedBy = GrossNotLessThanNetValidator.class)
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface GrossNotLessThanNet {
    String message() default "totalCostGross must be greater than or equal to totalCostNet";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}

public interface NetGrossHolder {
    BigDecimal totalCostNet();
    BigDecimal totalCostGross();
}

public class GrossNotLessThanNetValidator implements
ConstraintValidator<GrossNotLessThanNet, NetGrossHolder> {

    @Override
    public boolean isValid(NetGrossHolder value, ConstraintValidatorContext
context) {
        if (value == null) return true;
        BigDecimal net = value.totalCostNet();
        BigDecimal gross = value.totalCostGross();
        if (net == null || gross == null) return true;
        return gross.compareTo(net) >= 0;
    }
}

```

Globalna obsługa błędów

Centralnym punktem obsługi błędów jest klasa `GlobalExceptionHandler`, oznaczona adnotacją `@RestControllerAdvice`, którą prezentuje Listing 21. Przechwytuje ona wyjątki zgłoszone w dowolnym kontrolerze danego mikroserwisu i odwzorowuje je na odpowiednie kody statusu HTTP wraz z ujednoliconym ciałem odpowiedzi (znacznik czasu, status, opis błędu). Pozwala to uniknąć powielania obsługi wyjątków w kontrolerach i utrzymać spójny kontrakt błędów. Zdefiniowano m.in. następujące błędy:

- niepowodzenie walidacji (`MethodArgumentNotValidException`) – 400 Bad Request, wraz z listą pól i komunikatów,
- błędnie sformatowane żądanie / niepoprawny JSON (`HttpMessageNotReadableException`) – 400 Bad Request,
- brak zasobu (`IllegalArgumentException`, `EntityNotFoundException`) – 404 Not Found,
- naruszenie reguły biznesowej lub konflikt stanu (`IllegalStateException`) – 409 Conflict,
- nieobsługiwana metoda lub typ treści – 405 Conflict / 415 Unsupported Media Type,
- pozostałe, nieprzewidziane wyjątki – 500 Internal Server Error (z zapisem pełnego ładu stosu do dziennika).

Listing 21 Skrócony fragment globalnej obsługi wyjątków (`@RestControllerAdvice`) z mapowaniem wyjątków na kody HTTP. Źródło: Opracowanie własne.

```
@RestControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public Map<String, Object> handleValidation(MethodArgumentNotValidException e) {
        log.error("An error occurred while processing request, returning HTTP 400 Bad Request. Error {}", e.getMessage());
        return Map.of(
            "timestamp", OffsetDateTime.now(),
            "status", 400,
            "errors", "Validation failed",
            "details", e.getBindingResult().getFieldErrors().stream()
                .map(error -> Map.of("field", error.getField(),
"message", error.getDefaultMessage()))
                .toList()
        );
    }

    @ExceptionHandler(IllegalArgumentException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public Map<String, Object> handleNotFound(IllegalArgumentException e) {
        log.error("An error occurred while processing request, returning HTTP 404 Not Found. Error {}", e.getMessage());
        return Map.of(
            "timestamp", OffsetDateTime.now(),
            "status", 404,
            "error", e.getMessage()
        );
    }
}
```

```

    );
}

@ExceptionHandler(IllegalStateException.class)
@ResponseStatus(HttpStatus.CONFLICT)
public Map<String, Object> handleConflict(IllegalStateException e) {
    log.error("An error occurred while processing request, returning HTTP 409 Conflict. Error {}", e.getMessage());
    return Map.of(
        "timestamp", OffsetDateTime.now(),
        "status", 409,
        "error", e.getMessage()
    );
}

@ExceptionHandler(HttpMediaTypeNotSupportedException.class)
@ResponseStatus(HttpStatus.UNSUPPORTED_MEDIA_TYPE)
public Map<String, Object>
handleUnsupportedMediaType(HttpMediaTypeNotSupportedException e) {
    log.error("An error occurred while processing request, returning HTTP 415 Unsupported media type. Error {}", e.getMessage());
    return Map.of(
        "timestamp", OffsetDateTime.now(),
        "status", 415,
        "error", "Unsupported Media Type",
        "details", "Unsupported Media Type - must be application/json"
    );
}

@ExceptionHandler(Exception.class)
@ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
public Map<String, Object> handleException(Exception e) {
    log.error("An error occurred while processing request, returning HTTP 500 Internal Server error. Error {}", e.getMessage());
    log.error("Full stack trace: {}", Arrays.toString(e.getStackTrace()));
    return Map.of(
        "timestamp", OffsetDateTime.now(),
        "status", 500,
        "error", "Internal Server Error",
        "details", e.getMessage()
    );
}
}

```

6.2.5 Konfiguracja brokera RabbitMQ i komunikacja oparta na zdarzeniach

Konfiguracja

Jednym z podstawowych założeń pracy było to, żeby komunikacja między mikroserwisami była asynchroniczna i zorientowana na zdarzenia. W związku z tym, topologię tych wiadomości oparto na brokerze RabbitMQ i dwóch wymiennikach (ang. *exchanges*) typu topic. Ich strukturę oraz argumentację wyboru przybliżono w rozdziale 5.2.

Statyczna konfiguracja kolejek i wymienników brokera ładowana jest na podstawie pliku `definitions.json`, którego część została przedstawiona na Rysunek 40.

```

1      "exchanges": [
2      {
3          "name": "domain.rpc",
4          "vhost": "/",
5          "type": "topic",
6          "durable": true,
7          "auto_delete": false,
8          "arguments": {}
9      },
10     {
11         "name": "domain.events",
12         "vhost": "/",
13         "type": "topic",
14         "durable": true,
15         "auto_delete": false,
16         "arguments": {}
17     }
18 ],
19     "queues": [
20     {
21         "name": "asset.rpc.queue",
22         "vhost": "/",
23         "durable": true,
24         "auto_delete": false,
25         "arguments": {}
26     },
27     {
28         "name": "loan.rpc.queue",
29         "vhost": "/",
30         "durable": true,
31         "auto_delete": false,
32         "arguments": {}
33     },

```

Rysunek 40 Wycinek pliku konfiguracyjnego brokera RabbitMQ. Źródło: Opracowanie własne.

Kolejki konsumenckie zdarzeń deklarowane są natomiast dynamicznie, po stronie aplikacji, przy użyciu komponentów `@Bean` typu `Queue`, `Exchange` i `Binding` frameworka Spring AMQP - dzięki czemu każdy serwis samodzielnie ogłasza, jakie zdarzenia chce odbierać. Wspólną konfigurację brokera w pojedynczym serwisie przedstawiono na Listingu 22.

Listing 22 Dynamiczna konfiguracja Beanów typu Queue, Binding oraz Exchange w pliku konfiguracyjnym do klienta RabbitMQ. Źródło: Opracowanie własne.

```
@Configuration
public class Rabbit {
    private static final String RPC_EXCHANGE_NAME = "domain.rpc";
    private static final String EVENT_EXCHANGE_NAME = "domain.events";
    private static final String ASSET_CAMPAIGN_QUEUE = "asset-
campaign.events.queue";

    @Bean
    public Exchange rpcExchange() {
        return new TopicExchange(RPC_EXCHANGE_NAME, true, false);
    }

    @Bean
    public Exchange eventExchange() {
        return new TopicExchange(EVENT_EXCHANGE_NAME, true, false);
    }

    @Bean
    public DefaultClassMapper classMapper() {
        DefaultClassMapper classMapper = new DefaultClassMapper();
        classMapper.setTrustedPackages("*");
        return classMapper;
    }

    @Bean
    public JacksonJsonMessageConverter jsonMessageConverter() {
        JacksonJsonMessageConverter converter = new
JacksonJsonMessageConverter();
        converter.setClassMapper(classMapper());
        return converter;
    }

    @Bean
    public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory,
MessageConverter messageConverter) {
        RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
        rabbitTemplate.setMessageConverter(messageConverter);
        rabbitTemplate.setReplyTimeout(10000);
        return rabbitTemplate;
    }

    @Bean
    public SimpleRabbitListenerContainerFactory
listenerContainerFactory(ConnectionFactory connectionFactory, MessageConverter
messageConverter) {
        SimpleRabbitListenerContainerFactory factory = new
SimpleRabbitListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory);
        factory.setMessageConverter(messageConverter);
        factory.setDefaultRequeueRejected(false);
        return factory;
    }
}
```

```

@Bean
public Queue assetCampaignEventQueue() {
    return new Queue(ASSET_CAMPAIGN_QUEUE, true);
}

@Bean
public Binding campaignStartedBinding(Queue assetCampaignEventQueue, Exchange
eventExchange) {
    return BindingBuilder
        .bind(assetCampaignEventQueue)
        .to(eventExchange)
        .with("event.campaign.started").noargs();
}

```

Komunikaty serializowane są do formatu JSON przy pomocy konwertera JacksonJsonMessageConverter, uzupełnionego o DefaultClassMapper odpowiedzialnego za mapowanie typów. Kolejki oraz wymiennik są trwałe. Oznacza to, że podczas awarii brokera, gdy dojdzie do jego restartu, wiadomości jeszcze nie odebrane nie zostaną usunięte.

Publikacja zdarzeń

Publikacja zdarzeń odbywa się zgodnie z regułą opisaną w podrozdziale 5.5, co gwarantuje spójność między stanem bazy danych a komunikatem wysłanym do brokera. Mechanizm ten polega na tym, że serwis nie publikuje zdarzenia bezpośrednio do RabbitMQ, lecz publikuje wewnętrzne zdarzenie aplikacyjne Spring (ApplicationEventPublisher), natomiast dedykowany komponent reaguje na nie metodą oznaczoną adnotacją TransactionalEventListener w fazie AFTER_COMMIT.

W konsekwencji, komunikat trafia do brokera dopiero po zatwierdzeniu transakcji bazodanowej. Stanowi to praktyczną realizację wzorca Obserwatora, co przedstawiono i omówiono na przykładzie w podrozdziale 6.1.4, wraz z Listingiem 7, prezentującym sposób nadawania komunikatów.

Konsumpcja zdarzeń

Odbiór komunikatów realizowany jest przez komponenty oznaczone adnotacją @RabbitListener wskazującą nasłuchiwaną kolejkę, w których poszczególne metody @RabbitHandler dobierane są automatycznie na podstawie typu klasy. Struktura przepływu ma pewne podobieństwa do struktury kontrolerów i serwisów RESTowych; w szczególności, w kontrolerze zdefiniowane są wyżej wymienione metody nasłuchujące, w serwisie zaś metody obsługujące; przykład kontrolera typu RPC, nasłuchującego na kolejce asset.rpc.queue zademonstrowano na Listingu 23 serwisu zaś na Listingu 24.

Listing 23 Kontroler AssetRPCController. Źródło: Opracowanie własne.

```

@Service
@RabbitListener(queues = "asset.rpc.queue")
@RequiredArgsConstructor
public class AssetRPCController {
    private final AssetService assetService;
    @RabbitHandler
    public AssetResponses.AssetsDTO
getAssetsByLocationIds(AssetRequests.GetAssetsByLocationIdsRequest request) {
        return assetService.getAssetsByLocationIds(request);
    }

    @RabbitHandler

```

```

    public void patchAssetAsAssigned(AssetRequests.PatchAssetAsAssignedRequest
request) {
        assetService.patchAssetAsAssigned(request);
    }

    @RabbitHandler
    public void
patchAssetAsUnassigned(AssetRequests.PatchAssetAsUnassignedRequest request) {
        assetService.patchAssetAsUnassigned(request);
    }
}

```

Listing 24 Metoda zmieniająca status środka trwałego w konsekwencji otrzymanego żądania typu RPC.

Źródło: Opracowanie własne.

```

@Transactional
public AssetResponses.AssetDTO
patchAssetAsUnassigned(AssetRequests.PatchAssetAsUnassignedRequest request) {
    log.info("[RPC] Patching asset as unassigned: {}", request.assetId());

    Asset asset = assetRepo.findById(request.assetId())
        .orElseThrow(() -> new IllegalArgumentException("Asset not found:
" + request.assetId()));
    if (asset.getDuringInventory()) {
        throw new IllegalStateException("Asset is during inventory");
    }
    asset.setAssigned(false);
    Asset updatedAsset = assetRepo.save(asset);

    log.info("[RPC] Asset patched as unassigned: {}", request.assetId());

    return mapper.toRPCResponse(updatedAsset);
}

```

Wspólny kontrakt API

Aby zapewnić spójność co do formy i typu komunikacji, zdecydowano się na wydzielenie wszystkich żądań i odpowiedzi do wspólnej biblioteki `ams-shared-api`, dołączonej jako zależność do każdego mikroservisu. Kontrakty zdefiniowano przy użyciu Java Records, implementujących interfejs `Serializable`, pogrupowanych według dziedzin (np. `AssetRequests`, `AssetResponses` itd.). Przykład kontraktów żądań zaprezentowano na Listingu 25, zaś odpowiedzi na Listingu 26.

Listing 25 Współdzielony kontrakt żądań `AssetRequests`. **Źródło:** Opracowanie własne.

```

package com.pjatk.ams.amssharedapi.api.asset;

public interface AssetRequests {

    //      [...]
    public record GetAssetByIdRequest(java.util.UUID assetId) { }
    public record GetAssetsByIdsRequest(java.util.Set<java.util.UUID>
assetIds) { }
    public record GetAssetsByLocationIdRequest(java.util.UUID locationId) { }
    public record GetAssetsByLocationIdsRequest(java.util.Set<java.util.UUID>
locationIds) { }
}

```

```
[..]
    public record PatchAssetResponsibleUnitRequest(java.util.UUID assetId,
java.util.UUID responsibleUnitId) { }
    public record PatchAssetClearResponsibleUnitRequest(java.util.UUID
assetId) { }
    // [..]
}
```

Listing 26 Współdzielony kontrakt odpowiedzi AssetResponses. Źródło: Opracowanie własne.

```
package com.pjatk.ams.amssharedapi.api.asset;

public interface AssetResponses {
    public record AssetsDTO(
        java.util.Set<AssetDTO> assets
    ) { }

    public record AssetDTO(
        java.util.UUID id,
        String assetTag,
        String name,
        String description,
        java.util.UUID responsibleUnitUuid,
        String assetCondition,
        java.time.LocalDateTime lastCampaignDate,
        java.util.Map<String, Object> customData,
        java.util.UUID typeUuid,
        java.util.UUID locationUuid,
        java.util.UUID purchaseUuid,
        java.util.UUID categoryUuid,
        String vatRateName,
        java.math.BigDecimal unitPriceNet,
        java.math.BigDecimal unitPriceGross,
        String currency,
        Boolean assigned,
        Boolean duringInventory
    ) { }

    public record LocationsDTO(java.util.Set<LocationDTO> locations) { }

    public record LocationDTO(
        java.util.UUID id,
        String building,
        Integer floor,
        String room,
        String description
    ) { }
}
```

Istnieje jeszcze jeden dodatkowy powód zastosowania współdzielonych kontraktów; domyślnie, mechanizm rozpoznawania typu obiektów, o którym wspomniano wyżej w podrozdziale w odniesieniu do adnotacji `DefaultClassMapper`, działa na zasadzie pełnej ścieżki pakietu danej klasy. Oznacza to, że po obu stronach komunikacji, absolutna ścieżka do kontraktu musi być identyczna. W przeciwnym wypadku, konsument nie będzie w stanie zdeserializować wiadomości, gdyż nie będzie miał dostępu do odpowiedniej klasy.

6.3. Implementacja klienckiej aplikacji webowej – frontend

Warstwa kliencka systemu Glowing Waffle AMS zbudowana jest w oparciu o framework Next.js z językiem TypeScript, z wykorzystaniem modelu *App Router*. Stylizację interfejsu oparto na frameworku Tailwind CSS, uzupełnionym o bibliotekę komponentów Shadcn/UI, która dostarcza gotowe, edytowalne elementy interfejsu użytkownika. Komunikacja z mikroservisami backendowymi odbywa się poprzez dedykowaną warstwę pośredniczącą, a stan dziedzinowy aplikacji zarządzany jest przy pomocy kontekstów Reacta. Integracja z serwerem Keycloak zrealizowana jest przy użyciu biblioteki next-auth, odpowiedzialnej za obsługę sesji i tokenów JWT po stronie klienta.

6.3.1 Stylizacja i system komponentów UI

Stylizacja interfejsu użytkownika oparta jest na frameworku Tailwind CSS w wersji 4, który zastępuje tradycyjne pliki CSS zestawem klas użytkowych (ang. *utility classes*) stosowanych bezpośrednio w kodzie komponentów. Podejście to eliminuje konieczność tworzenia i utrzymywania osobnych arkuszy stylów, a każda klasa odpowiada pojedynczej właściwości CSS. Przykładowo, klasa `flex items-center justify-center h-screen` odpowiada regułom CSS `display: flex; align-items: center; justify-content: center; height: 100vh`. Paleta kolorów, zaokrąglenia narożników oraz typografia zdefiniowane są jako zmienne CSS w pliku `globals.css`, co pozwala na spójną zmianę wyglądu całej aplikacji w jednym miejscu. Listing 27 przedstawia fragment definicji zmiennej kolorystycznej.

Listing 27 Fragment definicji zmiennych kolorystycznych w pliku `globals.css`. Źródło: Opracowanie własne.

```
:root {
  --radius: 0.625rem;
  --background: oklch(1 0 0);
  --foreground: oklch(0.145 0 0);
  --primary: oklch(0.205 0 0);
  --primary-foreground: oklch(0.985 0 0);
  --muted: oklch(0.97 0 0);
  --muted-foreground: oklch(0.556 0 0);
}
```

Gotowe komponenty interfejsu użytkownika dostarczane są poprzez bibliotekę Shadcn/UI, które jest rozwiązaniem analogicznych do Material UI, lecz opartym na odmiennym modelu dystrybucji. Zamiast instalowania gotowej biblioteki jako zależności projektu, komponenty Shadcn/UI są kopiowane bezpośrednio do katalogu `components/ui/` projektu jako edytowalny kod źródłowy. Podejście to daje pełną kontrolę nad implementacją każdego komponentu. Pod spodem każdy komponent Shadcn/UI korzysta z biblioteki Radix UI, dostarczającej dostępne semantycznie (ang. *accessible*) prymitywne interfejsy, oraz biblioteki class-variance-authority (CVA) do zarządzania wariantami stylów. Listing 28 przedstawia fragment implementacji komponentu Button, ilustrujący ten mechanizm.

Listing 28 Fragment implementacji komponentu Button z użyciem biblioteki class-variance-authority. Źródło: Opracowanie własne.

```
const buttonVariants = cva(
  "inline-flex items-center justify-center gap-2 whitespace-nowrap rounded-md text-sm font-medium transition-all",
  {
    variants: {
      variant: {
        default: "bg-primary text-primary-foreground hover:bg-primary/90",
        destructive:
          "bg-destructive text-white hover:bg-destructive/90",
```

```

outline:
  "border bg-background shadow-xs hover:bg-accent",
secondary:
  "bg-secondary text-secondary-foreground",
ghost:
  "hover:bg-accent hover:text-accent-foreground",
},
size: {
  default: "h-9 px-4 py-2",
  sm: "h-8 rounded-md gap-1.5 px-3",
  lg: "h-10 rounded-md px-6",
  icon: "size-9",
},
}
};

```

Spalającym elementem między Tailwind CSS a komponentami Shadcn/UI jest funkcja pomocnicza `cn`, zdefiniowana w pliku `lib/utls.ts`. Łączy ona biblioteki `clsx` i `tailwind-merge`, co pozwala na bezpieczne scalanie klas Tailwind bez konfliktów. Listing 29 prezentuje jej implementację.

Listing 29 Implementacja funkcji `cn` łączącej biblioteki `clsx` i `tailwind-merge`. Źródło: Opracowanie własne.

```

import { clsx, type ClassValue } from "clsx"
import { twMerge } from "tailwind-merge"

export function cn(...inputs: ClassValue[]) {
  return twMerge(clsx(inputs))
}

```

Ikony w całej aplikacji pochodzą z biblioteki `Lucide React`, oferującej spójny, liniowy zestaw ponad 1000 ikon dostępnych jako komponenty `React`. W projekcie ikony używane są m.in. w bocznym pasku nawigacyjnym, przyciskach akcji oraz wskaźnikach stanu środków trwałych.

6.3.2 Struktura projektu

Projekt frontendu zorganizowany jest według konwencji *App Router* frameworka `Next.js`. Listing 30 przedstawia uproszczone drzewo katalogów projektu z opisem przeznaczenia każdego z nich.

Listing 30 Uprozczone drzewo katalogów projektu klienckiego. Źródło: Opracowanie własne.

```

frontend-service/
├─ app/
│  ├─ api/                                # Warstwa pośrednicząca Route Handlers
│  │  ├─ apiClient.                      # Wspólna funkcja komunikacji z mikroserwisami
│  │  ├─ asset-service/                  # Handler dla Asset Service
│  │  ├─ incident-service/               # Handler dla Incident Service
│  │  ├─ inventory-service/              # Handler dla Inventory Service
│  │  ├─ loan-service/                   # Handler dla Loan Service
│  │  ├─ kc-proxy/                       # Handler dla KC Proxy
│  │  └─ auth/                           # Konfiguracja next-auth i odświeżanie tokenu
│  ├─ assets-flow/                       # Widoki ewidencji środków trwałych
│  │  ├─ assets/[id]/                   # Szczegóły środka trwałego (segment dynamiczny)
│  │  ├─ categories/                    # Zarządzanie kategoriami
│  │  └─ locations/                     # Zarządzanie lokalizacjami
│  ├─ campaigns-flow/                   # Widoki kampanii inwentaryzacyjnych
│  ├─ incidents-flow/                   # Widoki incydentów
│  └─ loans-flow/                       # Widoki wypożyczeń

```

		personal/	# Widoki osobiste zalogowanego użytkownika
		context/	# Konteksty zarządzające stanem dziedzinowym
		ui/	# Współdzielone komponenty interfejsu
		layout.tsx	# Główny layout aplikacji
		└ providers.tsx	# Dostawcy kontekstu sesji i ochrona tras
		components/ui/	# Komponenty Shadcn/UI
		lib/	
		└ definitions.ts	# Definicje typów TypeScript
		└ utils.ts	# Funkcje pomocnicze
		hooks/	# Własne hooki Reacta
		└ public/	# Zasoby statyczne

Katalogi obszarów funkcjonalnych posiadają przyrostek *-flow (np. assets-flow/, capmains-flow/, incidents-flow/), co pozwala na jednoznaczne odróżnienie ich od pozostałych katalogów systemowych frameworka. Katalogi dynamicznych segmentów trasy, ujęte w nawiasy kwadratowe (np. assets/[id]/), przyjmują wartości identyfikatora zasobu z adresu URL i przekazują ją do komponentu strony jako parametr. Widoki osobiste użytkownika – niezwiązane z rolami operatorskimi – wyodrębniono do osobnego katalogu personal/, co odzwierciedla podział ról opisany w podrozdziale 3.4.

Kluczową rolę w inicjalizacji aplikacji pełni plik providers.tsx, który opakowuje drzewo komponentów dostawcami kontekstu. Oprócz dostarczania SessionProvider z biblioteki next-auth, plik ten definiuje komponent SessionGuard, odpowiedzialny za ochronę tras przed nieuwierzytelnionym dostępem. Listing 31 przedstawia jego implementację.

Listing 31 Implementacja komponentu SessionGuard chroniącego trasy przed nieuwierzytelnionym dostępem. Źródło: Opracowanie własne.

```
export function SessionGuard({ children }: { children: ReactNode }) {
  const { status, data: session } = useSession();
  const router = useRouter();
  const pathname = usePathname();

  useEffect(() => {
    if (status === "authenticated" && pathname === "/" ) {
      router.replace(getDefaultHomePath(session?.roles));
    }
  }, [status, pathname, router, session]);

  useEffect(() => {
    if (status === "unauthenticated" && pathname !== "/" ) {
      router.replace("/");
    }
  }, [status, pathname, router]);

  if (status === "loading") {
    return (
      <div>
        <span className="loader">
          <img src={icon.src} />
        </span>
        <span>Loading...</span>
      </div>
    );
  }
  return <>{children}</>;
}
```

```
}
```

Komponent reaguje na trzy stany sesji – loading, authenticated, unauthenticated. W stanie ładowania renderowany jest ekran oczekiwania, nieuwierzytlniony użytkownik jest przekierowywany na stronę logowania, a uwierzytlniony na widok domyślny zależny od jego roli.

6.3.3 Definicje typów TypeScript

Wszystkie typy danych wymienianych między frontendem a mikroserwisami backendowymi zdefiniowane są w pliku lib/definitions.ts. Podejście to zapewnia spójność kontraktów danych w całej aplikacji i umożliwia wykrywanie niezgodności typów już na etapie kompilacji, bez konieczności uruchamiania aplikacji. Listing 32 przedstawia definicję typu Asset, która stanowi centralną encję systemu, wraz z enumami Condition i Currency.

Listing 32 Definicja typu Asset wraz z enumami Condition i Currency. Źródło: Opracowanie własne.

```
export enum Condition {
  GOOD = "GOOD",
  DISCARDED = "DISCARDED",
  NEEDS_REPAIR = "NEEDS_REPAIR",
  TO_BE_DISCARDED = "TO_BE_DISCARDED",
  UNKNOWN = "UNKNOWN",
}

export enum Currency {
  PLN = "PLN",
  EUR = "EUR",
  USD = "USD",
  GBP = "GBP",
  CHF = "CHF",
}

export type Asset = {
  id: string;
  assetTag: string;
  name: string;
  description: string;
  responsibleUnitUuid: string;
  assetCondition: Condition;
  lastCampaignDate: string;
  typeUuid: string | null;
  locationUuid: string;
  purchaseUuid: string;
  categoryUuid: string;
  vatRateUuid: string;
  vatRateName: string;
  unitPriceNet: number;
  unitPriceGross: number;
  currency: Currency;
  assigned: boolean | number;
  customData?: Record<string, string | number | boolean | null>;
  duringInventory?: boolean;
};
```

Zastosowanie enumów dla pól przyjmujących skończony zbiór wartości, takich jak stan techniczny środka (Condition) czy waluta (Currency), eliminuje możliwość przekazania

nieprawidłowej wartości – kompilator TypeScript odrzuca wszelkie przypisania spoza zdefiniowanego zbioru. Warto zwrócić uwagę na pole `customData`, zdefiniowane jako słownik o wartościach typów prymitywnych. Odzwierciedla ono elastyczny model pól niestandardowych, który pozwala na bezpieczne typowo przechowywanie dowolnych pól opisowych zdefiniowanych przez administratora dla danej kategorii lub typu środka trwałego, bez konieczności modyfikacji schematu bazy danych.

Dla operacji wyszukiwania zdefiniowany jest osobny typ `AssetSearchRequest`, którego pola są w całości opcjonalne. Pozwala to na konstruowanie dowolnej kombinacji filtrów wyszukiwania bez konieczności tworzenia osobnych typów dla każdego wariantu zapytania. Listing 33 przedstawia jego definicję.

Listing 33 Definicja typu `AssetSearchRequest` reprezentującego kryteria wyszukiwania środków trwałych.
Źródło: Opracowanie własne.

```
export type AssetSearchRequest = {
  assetConditions?: string[] | null;
  assetTags?: string[] | null;
  assigned?: boolean | null;
  name?: string | null;
  responsibleUnitUids?: string[] | null;
  unitPriceFrom?: number | null;
  unitPriceTo?: number | null;
  lastCampaignDateFrom?: string | null;
  lastCampaignDateTo?: string | null;
  locationUids?: string[];
  assetCategoryUids?: string[];
  assetTypeUids?: string[];
  purchaseDateFrom?: string | null;
  purchaseDateTo?: string | null;
  purchaseUids?: string[] | null;
  duringInventory?: boolean;
};
```

6.3.4 Komunikacja z mikroservisami backendowymi

Całość komunikacji z mikroservisami odbywa się przez warstwę *Route Handlers* zgrupowanych w katalogu `/app/api/`. Dla każdego mikroservisu wyodrębniono osobny podkatalog (np. `assets-service/`, `incydent-service/`), w obrębie którego zdefiniowane są pliki `apiCalls.ts` zawierające funkcje wywołujące poszczególne zasoby REST. Taka organizacja izoluje logikę komunikacji od komponentów widoku i ukrywa przed przeglądarką adresy wewnętrzne mikroservisów.

Wspólna logika każdego żądania – pobieranie tokenu sesji, dołączanie nagłówka autoryzacyjnego oraz propagacja kontekstu śledzenia OpenTelemetry – scentralizowana jest w funkcji `apiClient`. Listing 34 przedstawia przykład fragmentu pliku `apiCalls.ts` dla mikroservisu środków trwałych, ilustrujący jak poszczególne operacje CRUD (Create, Read, Update, Delete) korzystają z tej funkcji.

Listing 34 Fragment pliku `apiCalls.ts` dla mikroservisu środków trwałych ilustrujący wzorzec wywołań CRUD przed funkcją `apiClient`.
Źródło: Opracowanie własne.

```
'use server'

import { apiClient, assetBase } from "@app/api/apiClient";
import { Asset } from "@lib/definitions";

export async function assets_get_all(): Promise<Asset[]> {
  return apiClient<Asset[]>(assetBase, "/asset-service", "/assets");
}
```

```

}

export async function assets_get(id: string): Promise<Asset> {
  return apiClient<Asset>(assetBase, "/asset-service", `/assets/${id}`);
}

export async function assets_delete(id: string): Promise<void> {
  return apiClient<void>(assetBase, "/asset-service", `/assets/${id}`, {
    method: "DELETE",
  });
}

export async function assets_post(payload: Partial<Asset>): Promise<Asset> {
  return apiClient<Asset>(assetBase, "/asset-service", "/assets", {
    method: "POST",
    body: JSON.stringify(payload),
  });
}

export async function assets_put(payload: Partial<Asset>, id: string):
Promise<Asset> {
  return apiClient<Asset>(assetBase, '/asset-service', `/assets/${id}`, {
    method: "PUT",
    body: JSON.stringify(payload),
  });
}

```

Dyrektywa `'use server'` na początku pliku oznacza, że wszystkie zawarte w nim funkcje są serwerowymi akcjami Next.js (ang. *Server Actions*), które są wykonywane wyłącznie po stronie serwera aplikacji, nigdy na przeglądarce. Dzięki temu token dostępu ani adresy wewnętrzne mikroserwisów nie są nigdy eksponowane po stronie klienta. Mechanizm odświeżania wygasającego tokenu dostępu zrealizowane zostało jako osobny Route Handler pod ścieżką `app/api/auth/refresh`. Listing 35 przedstawia jego implementację.

Listing 35 Implementacja Route Handlera odświeżającego token dostępu przez końcówkę Keycloak.
Źródło: Opracowanie własne.

```

export async function POST(req: NextRequest) {
  const token = await getToken({
    req,
    secret: process.env.NEXTAUTH_SECRET
  });

  if (!token?.refreshToken) {
    return NextResponse.json(
      { error: "No refresh token" },
      { status: 401 });
  }

  const params = new URLSearchParams();
  params.append("grant_type", "refresh_token");
  params.append("client_id", process.env.KEYCLOAK_CLIENT_ID!);
  params.append("client_secret", process.env.KEYCLOAK_CLIENT_SECRET!);
  params.append("refresh_token", token.refreshToken);

  const res = await fetch(`${process.env.KEYCLOAK_ISSUER}/protocol/openid-
connect/token`, {

```

```

    method: "POST",
    headers: { "Content-Type": "application/x-www-form-urlencoded" },
    body: params,
  });

  if (!res.ok) {
    return NextResponse.json(
      { error: "Cannot refresh token" },
      { status: 401 });
  }

  const data = await res.json();

  return NextResponse.json({
    accessToken: data.access_token,
    refreshToken: data.refresh_token ?? token.refreshToken,
    expiresAt: Math.floor(Date.now() / 1000) + data.expires_in,
  });
}

```

Handler pobiera token odświeżający z zaszyfrowanego ciasteczka sesji, a następnie wysyła żądanie do końcówki token serwera Keycloak z typem udziału (ang. *grant type*) ustawionym na `refresh_token`. W odpowiedzi zwraca nowy token dostępu, nowy token odświeżający oraz zaktualizowany czas wygaśnięcia, które komponent `TokenRefreshAlert` zapisuje do sesji po stronie klienta.

6.3.5 Zarządzaniem stanem aplikacji

Dane pobierane z mikroservisów przechowywane są w dedykowanych kontekstach Reacta (ang. *Context API*), zorganizowanych według obszarów dziedzicznych – `AssetsContext`, `InventoryContext`, `LoanContext`, `IncidentContext` oraz `KcProxyContext`. Każdy kontekst grupuje dane i operacje związane z jednym mikroservisem, udostępniając je wszystkim komponentom potomnym bez konieczności przez właściwości (ang. *props drilling*).

Interfejs każdego kontekstu definiuje dla każdego zasobu parę: listę pobranych danych wraz ze stanem ładowania oraz metodą `ensure*` inicjującą pobieranie. Listing 36 przedstawia fragment interfejsu `AssetContextType` przedstawiający ten wzorzec dla kilku zasobów.

Listing 36 Fragment interfejsu `AssetContextType` definiującego kontrakt kontekstu zasobów. Źródło: Opracowanie własne.

```

interface AssetContextType {
  assets: Asset[];
  assetsLoading: boolean;
  ensureAssets: () => Promise<void>;

  categories: AssetCategory[];
  categoriesLoading: boolean;
  ensureCategories: () => Promise<void>;

  locations: Location[];
  locationsLoading: boolean;
  ensureLocations: () => Promise<void>;

  getAsset: (id: string) => Promise<Asset>;
  addAsset: (data: Partial<Asset>) => Promise<Asset>;
}

```

```

updateAsset: (id: string, data: Partial<Asset>) => Promise<Asset>;
deleteAsset: (id: string) => Promise<void>;
}

```

Każdy metoda `ensure*` implementuje wzorec leniwego ładowania – sprawdza, czy dane zostały już pobrane, i inicjuje żądanie wyłącznie przy pierwszym zapotrzebowaniu, eliminując zbędne wywołania sieciowe przy kolejnych wejściach na widoki korzystające z tych samych danych. Listing 37 przedstawia ten mechanizm.

Listing 37 Implementacja mechanizmu leniwego ładowania w `AssetContext`. Źródło: Opracowanie własne.

```

const ensureAssets = async () => {
  if (assetsLoaded || assetsLoading) return;
  setAssetsLoading(true);
  try {
    await loadAssets()
    setAssetsLoaded(true);
  } finally {
    setAssetsLoading(false);
  }
};

```

Oprócz danych do odczytu, każdy kontekst udostępnia metody mutujące – `add*`, `update*` i `delete*` – które po wykonaniu operacji na backendzie aktualizują lokalny stan bez konieczności ponownego pobierania całej listy. Przykładowo po usunięciu kategorii środka trwałego wywoływane jest `setCategories((prev) => prev.filter(c => c.id !== id))`, co natychmiast odzwierciedla zmianę w interfejsie użytkownika.

6.3.6 Tabele danych z `TanStack Table`

Widoki prezentujące listy środków trwałych, kampanii, incydentów i wypożyczeń zbudowane są w oparciu o bibliotekę `TanStack Table`. Biblioteka ta dostarcza bezstanowy silnik tabelaryczny pozbawiony warstwy wizualnej – zarządza logiką sortowania, filtrowania, paginacji i widoczności kolumn, podczas gdy renderowanie pozostaje w gestii programisty. Takie podejście pozwala na pełną kontrolę nad wyglądem tabeli przy jednoczesnym korzystaniu z gotowej i przetestowanej logiki operacji na danych.

Każda tabela inicjalizowana jest przez hook `useReactTable`, który przyjmuje dane, definicje kolumn oraz zestaw aktywnych modeli funkcjonalnych. Listing 38 ilustruje konfigurację tabeli o uniwersalnym przeznaczeniu z włączonymi modelami sortowania, filtrowania i paginacji.

Listing 38 Inicjalizacja uniwersalnej tabeli przy użyciu hooka `useReactTable`. Źródło: Opracowanie własne.

```

const table = useReactTable({
  data,
  columns,
  meta,
  state: {
    sorting,
    columnFilters,
    globalFilter,
  },
  onSortingChange: setSorting,
  onColumnFiltersChange: setColumnFilters,
  onGlobalFilterChange: setGlobalFilter,
  getCoreRowModel: getCoreRowModel(),
});

```

```

getSortedRowModel: getSortedRowModel(),
getFilteredRowModel: getFilteredRowModel(),
getPaginationRowModel: getPaginationRowModel(),
});

```

Definicja kolumn przechowywana jest w osobnych plikach `*-columns.tsx` i opisuje każdą kolumnę obiektem `ColumnDef`, która zawiera klucz dostępu do dodanych, nagłówków oraz opcjonalny renderer komórki. Kolumny z możliwością sortowania przekazują do nagłówka funkcję przełączającą kierunek sortowania po kliknięciu, a kolumny prezentujące wartości zawarte w listach skończonych, takie jak stan techniczny środka, renderują komponent typu odznaki (ang. *badge*) z wariantem kolorystycznym odpowiadającym danej wartości.

Filtrowanie globalne zrealizowane jest w komponencie `TableFilters`, który posiada opóźnienie 300ms (ang. *debounce*) między wprowadzeniem znaku a faktycznym zastosowaniem filtra, zapobiegając nadmiernemu przerenderowaniu tabeli. Listing 39 przedstawia fragment implementacji komponentu filtrowania.

Listing 39 Fragment implementacji komponentu `TableFilters` z mechanizmem *debounce* i przełączaniem widoczności kolumn. Źródło: Opracowanie własne.

```

const [value, setValue] = useState(table.getState().globalFilter ?? "");

useEffect(() => {
  const id = setTimeout(() => {
    table.setGlobalFilter(value);
  }, 300);

  return () => clearTimeout(id);
}, [value, table]);

return (
  <div className="flex items-center py-4">
    <div className="relative">
      <Search/>
      <Input
        placeholder="Search by..."
        value={value}
        onChange={(e) => setValue(e.target.value)}
      />
    </div>
    {isDropdown && (
      <DropdownMenu>
        <DropdownMenuTrigger asChild>
          <Button variant="outline" className="ml-auto">
            Columns <ChevronDown />
          </Button>
        </DropdownMenuTrigger>
        <DropdownMenuContent align="end">
          {table.getAllColumns()
            .filter((column) => column.getCanHide())
            .map((column) => (
              <DropdownMenuCheckboxItem
                key={column.id}
                checked={column.getIsVisible()}
                onCheckedChange={(value) =>
                  column.toggleVisibility(!value)}
              > {column.id} </DropdownMenuCheckboxItem>
            ))}
        </DropdownMenuContent>
      </DropdownMenu>
    )}
  </div>
)

```

```

    </div>
  );
}

```

Komponent `TableFilters` udostępnia również menu rozwijane pozwalające na przełączanie widoczności poszczególnych kolumn – część kolumn jest domyślnie ukryta (np. `description`, `assetTag`), a użytkownik może je włączyć według potrzeb. Paginacja obsługiwana jest poprzez osobny komponent `PablePagination`, który renderuje przyciski nawigacji między stronami oraz selektor liczby wierszy wyświetlanych na stronę, z dostępnymi wartościami od 5 do 50, z przeskokiem co 5 lub 10.

6.3.7 Uwierzytelnienie i zarządzanie sesją

Integracja z serwerem Keycloak po stronie klienckiej zrealizowana jest przy użyciu biblioteki `next-auth` w wersji 4, skonfigurowanej z dostawcą `KeycloakProvider`. Dostawca przyjmuje trzy parametry środowiskowe: identyfikator klienta, sekret klienta oraz adres wydawcy tokenów (ang. *issuer*), co pozwala na zmianę serwera autoryzacji bez modyfikacji kodu źródłowego.

Biblioteka `next-auth` dostarcza dwa mechanizmy rozszerzania danych sesji – `callback jwt` oraz `callback session`. `Callback jwt` wywoływany jest przy każdym zdarzeniu dotyczącym tokenu i służy do dekodowania oraz utrwalania danych w zaszyfrowanym ciasteczku. `Callback session` wywoływany jest przy każdym odczycie sesji po stronie klienta i przepisuje wybrane dane z tokena do obiektu sesji dostępnego dla komponentów. Listing 40 przedstawia pełną konfigurację obu callbacków.

Listing 40 Konfiguracja callbacków jwt i session biblioteki next-auth. Źródło: Opracowanie własne.

```

callbacks: {
  async jwt({ token, account, profile }) {
    if (account) {
      token.accessToken = account.access_token;
      token.idToken = account.id_token;
      token.refreshToken = account.refresh_token;
      token.expiresAt = account.expires_at;
      if (account.access_token) {
        const decoded: any = jwtDecode(account.access_token);
        const clientRoles = decoded.resource_access?.["nextjs-app"]?.roles ||
[];

        token.roles = [...clientRoles];
      }
      token.id = account.providerAccountId;
      token.refreshToken = account.refresh_token;
      token.refreshExpiresIn = Number(Date.now().toString().slice(0, 10)) +
Number(account.refresh_expires_in);
    }
    if (token.expiresAt && Date.now() / 1000 > token.expiresAt) token.error =
"TokenExpired";
    return token;
  },
  async session({ session, token }) {
    session.accessToken = token.accessToken as string | undefined;
    session.roles = token.roles as string[] | undefined;
    session.expiresAt = token.expiresAt;
    session.error = token.error as string | undefined;
    session.refreshExpiresIn = token.refreshExpiresIn;
    session.user.id = token.id
    return session;
  },
}

```

W callbacku jwt token dostępu dekodowany jest przy użyciu biblioteki jwt-decode, skąd wyodrębniane są role przypisane klientowi nextjs-app w sekcji resource-access ładunku tokena. Role te dołączone są do obiektu sesji oraz stają się dostępne dla komponentów bez konieczności ponownego dekodowania tokena. Pole `refreshExpiresIn` przechowuje czas wygaśnięcia tokenu odświeżającego, który wykorzystywany jest przez komponent `TokenRefreshAlert` do dynamicznego obliczania progu ostrzeżenia o zbliżającym się końcu sesji, opisanego w podrozdziale 5.4.

Wylogowanie użytkownika propagowane jest do serwera Keycloak za pośrednictwem zdarzenia `signOut`. Listing 41 przedstawia implementację mechanizmu.

Listing 41 Implementacja zdarzenia `signOut` propagującego wylogowanie do serwera Keycloak. Źródło: Opracowanie własne.

```
events: {
  async signOut({ token }) {
    if (!token?.idToken) return;

    const issuer = process.env.KEYCLOAK_ISSUER!;
    const logoutUrl =
      `${issuer}/protocol/openid-connect/logout` +
      `?id_token_hint=${token.idToken}` +
      `&client_id=${process.env.KEYCLOAK_CLIENT_ID}`;

    await fetch(logoutUrl);
  },
},
```

Żądanie do końcówki `logout` Keycloaka zawiera parametr `id_token_hint` z tokenem identyfikującym bieżącą sesję, co umożliwia serwerowi autoryzacji jednoznaczną identyfikację sesji do unieważnienia. Dzięki temu wylogowanie po stronie klienta Next.js skutkuje równoczesnym zakończeniem sesji po stronie Keycloaka, bez możliwości dalszego użycia tokenu odświeżającego.

6.4. Mechanizmy zapewniające bezpieczeństwo

Podrozdział opisuje implementację mechanizmów bezpieczeństwa systemu Glowing Waffle AMS na obu warstwach – backendowej i frontendowej. Szersze decyzje architektoniczne dotyczące wyboru serwera autoryzacji Keycloak, przepływu *Authorization Code Flow* z PKCE oraz formatu tokenów JWT zostały omówione w podrozdziale 5.4. Niniejszy podrozdział skupia się wyłącznie na konkretnych rozwiązaniach implementacyjnych egzekwujących te decyzje w kodzie.

6.4.1 Katalog ról i moduł współdzielony

Centralnym elementem modelu uprawnień systemu jest klasa `AmsRole`, zdefiniowana w module `ams-shared-api` i importowana przez wszystkie mikroserwisy. Przechowuje ona nazwy ról jako stałe łańcuchy znaków zamiast wartości wyliczalnych – wynika to wymagania Spring Security, które oczekuje literałów łańcuchowych w wyrażeniach `hasRole`, `hasAnyRole` oraz adnotacjach `@PreAuthorize`. Listing 42 przedstawia pełną definicję katalogu ról.

Listing 42 Katalog ról systemu w klasie `AmsRole` modułu `ams-shared-api`. Źródło: Opracowanie własne.

```
public final class AmsRole {
  private AmsRole() {}

  public static final String ADMIN = "ams_admin";
```

```

public static final String USER = "ams_user";
public static final String ASSET_MANAGER = "asset_manager";
public static final String INVENTORY_AUDITOR = "inventory_auditor";
public static final String INVENTORY_COUNTER = "inventory_counter";
public static final String LOAN_APPROVER = "loan_approver";
public static final String INCIDENT_HANDLER = "incident_handler";
}

```

Nazwy ról muszą pozostawać zsynchronizowane z trzema miejscami w konfiguracji systemu: grupami LDAP zdefiniowanymi w pliku `base-structure.ldif`, rolami klienta `nextjs-app` w konfiguracji dziedziny Keycloak oraz enumem `Roles` po stronie frontendowej. Centralizacja nazw w jednym module eliminuje ryzyko rozbieżności między mikroserwisami wynikającej z literówek lub niespójnych nazewnictw.

6.4.2 Konfiguracja Spring Security i walidacja tokenów JWT

Każdy mikroserwis posiada własną klasę `SecurityConfig` konfigurującą łańcuch filtrów Spring Security. Wspólne dla wszystkich mikroserwisów są trzy ustawienia: wyłączenie mechanizmu CSRF (zbędne w architekturze bezstanowej opartej na tokenach JWT), wymuszenie bezstanowego zarządzania sesją (ang. *stateless session management*) oraz skonfigurowanie zasobów OAuth 2.0 weryfikującego tokeny JWT. Listing 43 przedstawia fragment konfiguracji dla Asset Service, ilustrujący granularny podział uprawnień na poziomie metod http i ścieżek URL.

Listing 43 Fragment konfiguracji Spring Security Asset Service z podziałem uprawnień według metod HTTP. Źródło: Opracowanie własne.

```

http.csrf(AbstractHttpConfigurer::disable)
.cors(Customizer.withDefaults())
.sessionManagement(session ->
    session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
.authorizeHttpRequests(auth -> auth
    .requestMatchers(HttpMethod.GET, "/api/v1/asset-service/assets/my")
        .authenticated()
    .requestMatchers(HttpMethod.POST, "/api/v1/asset-service/assets/search")
        .hasAnyRole(ADMIN, ASSET_MANAGER, INVENTORY_AUDITOR,
            LOAN_APPROVER, INCIDENT_HANDLER, USER)
    .requestMatchers(HttpMethod.GET, "/api/v1/asset-service/**")
        .hasAnyRole(ADMIN, ASSET_MANAGER, INVENTORY_AUDITOR,
            INVENTORY_COUNTER, LOAN_APPROVER,
            INCIDENT_HANDLER, USER)
    .requestMatchers(HttpMethod.POST, "/api/v1/asset-service/**")
        .hasAnyRole(ADMIN, ASSET_MANAGER)
    .requestMatchers(HttpMethod.PUT, "/api/v1/asset-service/**")
        .hasAnyRole(ADMIN, ASSET_MANAGER)
    .requestMatchers(HttpMethod.DELETE, "/api/v1/asset-service/**")
        .hasAnyRole(ADMIN, ASSET_MANAGER)
    .anyRequest().authenticated())
.oauth2ResourceServer(oauth2 ->
    oauth2.jwt(jwt ->
        jwt.jwtAuthenticationConverter(jwtAuthenticationConverter())));

```

Warto zwrócić uwagę na końcówkę `/assets/search` – semantycznie jest to operacja odczytu, jednak ze względu na złożoność kryteriów filtrowania zrealizowana jest jako żądanie POST z ciałem zawierającym parametry wyszukiwania. Wymaga to osobnego wpisu w konfiguracji, ponieważ ogólna reguła dla metody POST ogranicza dostęp wyłącznie do ról zarządzających.

6.4.3 Konwerter ról Keycloak i autoryzacja na poziomie metod

Tokeny JWT wydawane przez Keycloak przechowują role użytkownika w niestandardowej sekcji `resource_access` ładunku, a nie w standardowym polu `authorities`. Spring Security nie rozumie tej struktury natywnie, dlatego każdy mikroserwis rejestruje konwerter `KeycloakRoleConverter`, który wyodrębnia role z właściwości sekcji tokenu i mapuje je na obiekty `GrantedAuthority` z przedrostkiem `ROLE_` wymagany przez Spring Security. Listing 44 prezentuje jego implementację.

Listing 44 Implementacja `KeycloakRoleConverter` mapującego role Keycloak na uprawnienia Spring Security. Źródło: Opracowanie własne.

```
public Collection<GrantedAuthority> convert(Jwt jwt) {
    Map<String, Object> resourceAccess =
        jwt.getClaimAsMap("resource_access");
    if (resourceAccess == null || resourceAccess.isEmpty()) {
        return Collections.emptyList();
    }
    Map<String, Object> roleMap =
        (Map<String, Object>) resourceAccess.get("nextjs-app");
    Collection<String> roles = (List<String>) roleMap.get("roles");
    if (roles == null || roles.isEmpty()) {
        return Collections.emptyList();
    }
    return roles.stream()
        .map(role -> new SimpleGrantedAuthority("ROLE_" + role))
        .collect(Collectors.toList());
}
```

Weryfikacja roli użytkownika na poziomie łańcucha filtrów http jest pierwszą linią obrony, jednak niewystarczającą dla zasobów wymagających weryfikacji własności rekordu. Rola bazowa `ams_user` uprawnia użytkownika do tworzenia wniosków o wypożyczenie i zgłaszanie incydentów, ale dostęp do konkretnego rekordu powinien być ograniczony wyłącznie do jego własności. W tym celu we wszystkich mikroserwisach zdefiniowany jest komponent `SecurityComponent`, który wyodrębnia dane identyfikujące zalogowanego użytkownika bezpośrednio z kontekstu bezpieczeństwa Spring Security. Listing 45 prezentuje fragment tego mechanizmu.

Listing 45 Fragment implementacji komponentu `SecurityComponent` wyodrębniający dane użytkownika z tokenu JWT. Źródło: Opracowanie własne.

```
@Component
public class SecurityComponent {
    public UUID getCurrentUserId() {
        Authentication auth =
            SecurityContextHolder.getContext().getAuthentication();
        if (auth == null || !(auth.getPrincipal() instanceof Jwt jwt)) {
            throw new AccessDeniedException("No security context found - expected JWT");
        }
        try {
            return UUID.fromString(jwt.getSubject());
        } catch (IllegalArgumentException e) {
            throw new AccessDeniedException("Claim 'sub' from JWT is not valid UUID.", e);
        }
    }

    public String getCurrentUserEmail() {
        Authentication auth =
            SecurityContextHolder.getContext().getAuthentication();
    }
```

```

        if (auth == null || !(auth.getPrincipal() instanceof Jwt jwt)) {
            throw new AccessDeniedException("No security context found - expected
JWT");
        }
        return jwt.hasClaim("email") ? (String) jwt.getClaim("email") : null;
    }
}

```

Komponent odczytuje identyfikator użytkownika z roszczenia sub (subject) tokenu JWT – jest to UUID konta w Keycloak, które pozostaje niezmiennie przez cały cykl życia konta. W przypadku braku kontekstu bezpieczeństwa lub nieprawidłowego formatu identyfikatora rzucony jest wyjątek `AccessDeniedException`, który Spring Security przekształca w odpowiedź 403 Forbidden.

`SecurityComponent` stanowi podstawę dla dedykowanych komponentów autoryzacyjnych, eksponowanych do wyrażen SpEL (Spring Expression Language) przez adnotacje `@PreAuthorize`. Dla każdego mikroserwisu obsługującego zasoby z kontrolą własności zdefiniowano osobny bean autoryzacyjny, dla przykładu `IncidentAuthorization` dla incydentów oraz `LoanRequestAuthorization` dla wniosków wypożyczeń. Listing 46 przedstawia użycie mechanizmu w kontrolerze wniosków o wypożyczenie.

Listing 46 Zastosowanie adnotacji `@PreAuthorize` z weryfikacją własności rekordu w kontrolerze wniosków o wypożyczenie. Źródło: Opracowanie własne.

```

@GetMapping("/{uuid}")
@PreAuthorize("hasAnyRole('ams_admin', 'loan_approver', 'asset_manager') " +
    "or @loanRequestSecurity.isOwner(#uuid)")
public LoanRequestResponse getLoanRequestById(@PathVariable UUID uuid) {
    return loanRequestService.findByUuid(uuid);
}

@PutMapping("/{uuid}")
@PreAuthorize("hasAnyRole('ams_admin', 'loan_approver') " +
    "or @loanRequestSecurity.isOwner(#uuid)")
public LoanRequestResponse updateLoanRequest(
    @RequestBody @Valid LoanRequestUpdateRequest request,
    @PathVariable UUID uuid) {
    return loanRequestService.update(request, uuid);
}

@DeleteMapping("/{uuid}")
@PreAuthorize("hasRole('ams_admin') or @loanRequestSecurity.isOwner(#uuid)")
public void deleteLoanRequest(@PathVariable UUID uuid) {
    loanRequestService.deleteByUuid(uuid);
}

```

Bean `loanRequestSecurity` odpytuje repozytorium i porównuje identyfikator właściciela wniosku z identyfikatorem zalogowanego użytkownika pobranym przez `SecurityComponent`. W przypadku nieistniejącego rekordu metoda zwraca wartość `false`, co skutkuje odpowiedzią 403 Forbidden zamiast 404 Not Found – celowe zachowanie eliminujące możliwość wykrycia istnienia zasobu przez nieuprawnionego użytkownika (ang. *resource existence leakage*).

6.4.4 Konfiguracja CORS

Każdy mikroserwis definiuje własną politykę CORS (*Cross-Origin Resource Sharing*) jako bean `CorsConfigurationSource`. Mechanizm CORS jest wymuszany przez przeglądarki internetowe i służy ograniczeniu możliwości wykonywania żądań do API z domen innych niż dozwolone. Konfiguracja jest uzależniona od aktywnego profilu Spring – w profilu deweloperskim dozwolone są

żądania dowolnego pochodzenia, natomiast w środowisku produkcyjnym ograniczone zostały do konkretnych domen aplikacji frontendowej. Listing 47 przedstawia tę konfigurację.

Listing 47 Konfiguracja polityki CORS z rozróżnieniem środowiska deweloperskiego i produkcyjnego.

Źródło: Opracowanie własne.

```
@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();

    if (env.acceptsProfiles(Profiles.of("docker"))) {
        configuration.setAllowedOriginPatterns(List.of("*"));
    } else {
        configuration.setAllowedOrigins(List.of(
            "https://ams.glowingwaffle.app",
            "https://assets-app.glowingwaffle.app"
        ));
    }

    configuration.setAllowedMethods(
        List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));
    configuration.setAllowedHeaders(
        List.of("Authorization", "Content-Type"));

    UrlBasedCorsConfigurationSource source =
        new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/*", configuration);
    return source;
}
```

Jawne zdefiniowanie dozwolonych nagłówków (Authorization, Content-Type) zapobiega przekazywaniu przez przeglądarkę niestandardowych nagłówków, które mogłyby zostać wykorzystane do przeprowadzenia ataku. Nagłówek Authorization jest niezbędny do przekazywania tokenów JWT, natomiast Content-Type wymagany jest do prawidłowej interpretacji ciała żądań zawierających dane w formacie JSON.

6.4.5 Ochrona tras i warunkowe renderowanie po stronie aplikacji klienckiej

Po stronie frontendowej pierwszą warstwą ochrony jest middleware Next.js zdefiniowany w pliku proxy.ts, przechwytyjący każde żądanie nawigacyjne przed jego obsłużeniem przez aplikację. Middleware weryfikuje obecność tokenu sesji i kieruje nieuwierzytelnionych użytkowników na stronę logowania. Dla żądań od uwierzytelnionych użytkowników wykonywana jest introspekcja tokenu dostępu przez dedykowaną końcówkę Keycloak. Mechanizm ten adresuje opisaną w podrozdziale 5.4 wadę tokenów JWT polegającą na braku natywnego mechanizmu unieważnienia – nawet jeśli token jest jeszcze ważny według swojego czasu wygaśnięcia, introspekcja pozwala wykryć że został on unieważniony po stronie serwera autoryzacji. Listing 48 przedstawia pełną implementację funkcji introspekcji.

Listing 48 Implementacja funkcji isTokenValid wykonującej introspekcję tokenu przez końcówkę

Keycloak. Źródło: Opracowanie własne.

```
async function isTokenValid(accessToken: string) {
    const issuer = process.env.KEYCLOAK_ISSUER!;
    const introspectUrl = `${issuer}/protocol/openid-connect/token/introspect`;

    const params = new URLSearchParams();
    params.append("token", accessToken);
```

```

params.append("client_id", process.env.KEYCLOAK_CLIENT_ID!);
params.append("client_secret", process.env.KEYCLOAK_CLIENT_SECRET!);

try {
  const res = await fetch(introspectUrl, {
    method: "POST",
    headers: {
      "Content-Type": "application/x-www-form-urlencoded",
    },
    body: params,
  });

  if (!res.ok) return true;

  const data = await res.json();
  return data.active === true;
} catch (err) {
  return true;
}
}

```

Warto zwrócić uwagę na strategię obsługi błędów – zarówno nieudane żądanie do Keycloak, jak i wyjątek sieciowy skutkują zwróceniem wartości `true`, czyli oceną tokena jako ważnego. Jest to świadoma decyzja projektowa: chwilowa niedostępność serwera autoryzacji nie powinna blokować dostępu zalogowanym użytkownikom, co wpisuje się w zasadę odporności opisaną w podrozdziale 5.4.

Drugą warstwą ochrony po stronie klienckiej jest warunkowe renderowanie elementów interfejsu w zależności od ról zalogowanego użytkownika. Stosowana jest w tym celu funkcja pomocnicza `has`, sprawdzająca czy sesja zawiera co najmniej jedną z wymaganych ról. Listing 49 przedstawia jej definicję oraz reprezentatywny przykład użycia widoku szczegółów środka trwałego, gdzie ten sam komponent renderuje różny zestaw akcji i nawigacji w zależności od roli.

Listing 49 Warunkowe renderowanie elementów interfejsu w widoku szczegółów środka trwałego na poziomie ról użytkownika. Źródło: Opracowanie własne.

```

const has = (...roles: Roles[]) =>
  roles.some(role => session?.roles?.includes(role.toLowerCase()));

{has(Roles.ams_admin, Roles.asset_manager) ? (
  <p className="hover:underline hover:cursor-pointer"
    onClick={() => router.push(`/assets-flow/categories/${category.id}`)}>
    {category?.name || '-' }
  </p>
) : (
  <p>{category?.name || '-'}</p>
)}

{!has(Roles.loan_approver) && (
  <LoanSection loans={loans} requests={requests} />
)}

{!has(Roles.loan_approver) && (
  <CampaignSection campaigns={campaigns} />
)}

```

Podejście to realizuje zasadę najmniejszego uprawnienia (ang. *principle of least privilege*) na poziomie interfejsu – użytkownik widzi wyłącznie te akcje i sekcje, do których ma dostęp. Należy jednak podkreślić, że warunkowe renderowanie po stronie frontendu pełni wyłącznie rolę mechanizmu

poprawiającego ergonomię użytkownika, a nie bezpieczeństwa. Faktyczne egzekwowanie uprawnień odbywa się zawsze po stronie backendowej, w łańcuchu filtrów Spring Security i adnotacjach @PreAuthorize.

6.5. Konteneryzacja i orkiestracja środowiska – DevOps

Niniejszy podrozdział opisuje praktyczną realizację konteneryzacji oraz orkiestracji środowiska uruchomieniowego systemu Glowing Waffle AMS. Charakterystykę użytych narzędzi, czyli platformy Docker, bramy API Traefik oraz mechanizmu GitHub Actions, przedstawiono kolejno w podrozdziałach 4.9, 4.10 oraz 4.13, natomiast architekturę warstwy obserwowalności omówiono w podrozdziale 5.7. Tym samym poniższy opis koncentruje się wyłącznie na konkretnych rozwiązaniach: sposobie budowania obrazów kontenerowych, deklaratywnej orkiestracji całego środowiska, segmentacji sieci, routingu ruchu sieciowego oraz zautomatyzowanym potoku CI/CD.

W projekcie wyróżniono dwa równoległe środowiska. Środowisko deweloperskie (plik docker-compose.yaml) buduje obrazy lokalnie na podstawie kodu źródłowego, udostępnia porty poszczególnych usług na maszynie programisty i celowo upraszcza warstwę sieciową oraz zabezpieczeń, celem przyspieszenia procesu tworzenia kodu, a także ułatwieniu wyszukania bugów. Środowisko produkcyjne (plik docker-compose.prod.yaml) pobiera gotowe, opatrzone wersją obrazy z rejestru GitHub Registry, wprowadza segmentację sieci, terminację TLS w bramie API, kontrole stanu kontenerów oraz trwałe wolumeny danych.

6.5.1 Wieloetapowe budowanie obrazów kontenerów

Każdy mikroservis backendowy posiada własny plik Dockerfile, w którym zastosowano wieloetapowe budowanie obrazu (ang. *multi-stage build*). Podejście to rozdziela środowisko kompilacji od środowiska uruchomieniowego: pierwszy etap, oparty na pełnym obrazie z narzędziem Maven oraz pakietem JDK, kompiluje moduł współdzielony ams-shared-api (zawierający kontrakty metod API w formie Java Records, wykorzystywane do komunikacji przez brokera RabbitMQ) i kod mikroservisu do pliku wykonywalnego JAR, natomiast drugi etap kopiuje wyłącznie zbudowany artefakt do minimalnego obrazu bazowego. Listing 50 przedstawia plik Dockerfile mikroservisu środków trwałych *Asset Service*.

Listing 50 Wieloetapowy proces wytwarzania obrazu kontenera Asset Service w Dockerfile. Źródło: Opracowanie własne.

```
## Build stage
FROM maven:4.0.0-rc-4-eclipse-temurin-25 AS builder

WORKDIR /app

# OpenTelemetry Java Agent
RUN curl -sL -o /app/opentelemetry-javaagent.jar https://github.com/open-telemetry/opentelemetry-java-instrumentation/releases/download/v2.26.1/opentelemetry-javaagent.jar

# Shared API + Core microservice build
## Dependencies
COPY ams-shared-api/pom.xml ams-shared-api/
COPY asset-service/pom.xml asset-service/

COPY ams-shared-api/src/ ams-shared-api/src/
RUN mount=type=cache,target=/root/.m2,sharing=locked mvn -B -f ams-shared-api/pom.xml clean install
```

```

RUN--mount=type=cache,target=/root/.m2,sharing=locked mvn -B -f asset-
service/pom.xml dependency:go-offline

COPY asset-service/src/ asset-service/src/
RUN--mount=type=cache,target=/root/.m2,sharing=locked mvn -B -f asset-
service/pom.xml clean package -DskipTests

## Run stage
FROM gcr.io/distroless/java25-debian13:nonroot
LABEL authors="Piotr Natkaniec, Tomasz Gajewski, Kacper Barański"

WORKDIR /app

COPY--from=builder-chown=nonroot:nonroot /app/asset-service/target/asset-
service*.jar app.jar
COPY--from=builder-chown=nonroot:nonroot /app/opentelemetry-javaagent.jar otel-
java-agent.jar

USER nonroot
ENV JAVA_TOOL_OPTIONS="-XX:MaxRAMPercentage=75.0 \
  -XX:+DisableAttachMechanism \
  -Djdk.attach.allowAttachSelf=false"
ENTRYPOINT ["java", "-javaagent:/app/otel-java-agent.jar", "-jar", "app.jar"]

```

Zastosowane rozwiązanie realizuje kilka istotnych założeń. Po pierwsze, końcowy obraz kontenera oparto na obrazie *distroless* od Google, zawierającym wyłącznie środowisko uruchomieniowe języka Java, bez powłoki systemowej (tj. *bash*), menedżera pakietów typu apt, yum, dnf ani dodatkowych narzędzi – co istotnie zmniejsza powierzchnię ataku oraz rozmiar końcowego obrazu. Według [50], najmniejszy dostępny obraz źródłowy, „gcr.io/distroless/static-debian13”, waży zaledwie 2,09 megabajtów. Co więcej, aplikacja uruchamiana jest w kontenerze na koncie nieuprzywilejowanym (użytkownik nonroot), co jeszcze bardziej zmniejsza powierzchnię do potencjalnego ataku. Po trzecie, w zmiennej `JAVA_TOOL_OPTIONS` ograniczono maksymalne zużycie pamięci serty do 75% pamięci dostępnej dla kontenera, a także wyłączono mechanizm dołączania agentów Java w czasie działania (ang. *attach mechanism*), utrudniając ingerencję w działający proces. Dyrektywy – `mount=type=cache` wykorzystują mechanizm pamięci podręcznej systemu budowania, dzięki czemu pobrane zależności Maven są współdzielone między kolejnymi etapami, co skraca czas kompilacji. Agent OpenTelemetry pobierany jest w etapie budowania i dołączany do procesu w czasie startu (parametr `-javaagent`), co zapewnia automatyczną instrumentację mikroservisu bez modyfikacji jego kodu źródłowego.

Analogiczne podejście zastosowano dla aplikacji frontendowej, której proces budowania jest jednak nieco odmienny ze względu na ekosystem Node.js. Listing 51 przedstawia plik Dockerfile warstwy klienckiej.

Listing 51 Wieloetapowy proces wytwarzania obrazu kontenera Frontend Service w Dockerfile. Źródło: Opracowanie własne.

```

# --- Build stage ---
FROM node:24-alpine AS builder
WORKDIR /app
RUN corepack enable && corepack prepare pnpm@latest-activate

# Cache and dependencies
COPY frontend-service/package.json frontend-service/pnpm-lock.yaml ./
RUN pnpm install-frozen-lockfile

```

```

# App build
COPY frontend-service/ ./
RUN pnpm build

# --- Run stage ---
FROM gcr.io/distroless/nodejs24-debian13:nonroot
WORKDIR /app

LABEL authors="Piotr Natkaniec, Tomasz Gajewski, Kacper Barański"
ENV NODE_ENV=production

# Copy files
COPY--from=builder-chown=nonroot:nonroot /app/public ./public
COPY--from=builder-chown=nonroot:nonroot /app/.next/standalone ./
COPY--from=builder-chown=nonroot:nonroot /app/.next/static ./next/static
COPY--from=builder-chown=nonroot:nonroot /app/.env.production.local ./

USER nonroot
CMD ["server.js"]

```

W etapie budowania menedżer pakietów pnpm aktywowany jest poprzez mechanizm corepack, a instalacja zależności wykonywana jest z opcją – `frozen-lockfile`, która gwarantuje zgodność z zatwierdzonym plikiem blokady poszczególnych numerów wersji (ang. *lockfile*) i uniemożliwia jego niejawną modyfikację podczas budowania. Wykorzystano tryb standalone frameworka Next.js, który tworzy aplikację zawierającą wyłącznie pliki niezbędne do uruchomienia, wraz z absolutnie niezbędnymi zależnościami. Dzięki temu do obrazu uruchomieniowego - również opartego na obrazie distroless z kontem nonroot - kopiowane są jedynie katalogi public, standalone oraz static, bez pełnego drzewa node_modules; folder ten, w dużych projektach, jest zazwyczaj ogromnych rozmiarów [51].

6.5.2 Orkiestracja środowiska za pomocą mechanizmu Docker Compose

Całością środowiska zarządza się deklaratywnie przy pomocy narzędzia Docker Compose. W środowisku produkcyjnym poszczególne usługi nie są budowane lokalnie, lecz pobierane jako gotowe obrazy z rejestru GitHub Container Registry, z polityką `pull_policy: always` wymuszającą pobranie najnowszej wersji przy każdym wdrożeniu. Listing 52 przedstawia – w postaci skróconej – definicję mikroserwisu środków trwałych wraz z towarzyszącą mu bazą danych.

Listing 52 Skrócona definicja mikroserwisu oraz jego bazy danych w pliku docker-compose.prod.yaml.

Źródło: Opracowanie własne.

```

services:
  asset-service:
    image: ghcr.io/patkaniec22/glowing-waffle/asset-service:latest
    pull_policy: always
    environment:
      SPRING_PROFILES_ACTIVE: production
      SPRING_DATASOURCE_URL: jdbc:postgresql://asset-service-db:5432/asset-
service-db
    # [...] pozostałe zmienne środowiskowe
  networks:
    - backend
    - observability
    - internal-network
  depends_on:
    asset-service-db:
      condition: service_healthy

```

```

    rabbitmq:
      condition: service_started

asset-service-db:
  image: postgres:15.17-alpine
  container_name: asset-service-db
  restart: always
  command:
    - "postgres"
    - "-c"
    - "max_connections=100"
    - "-c"
    - "shared_buffers=256MB"
  # [...] dalsze parametry strojenia bazy
  healthcheck:
    test: [ "CMD-SHELL", "pg_isready -U asset-app -d asset-service-db" ]
    interval: 10s
    timeout: 5s
    retries: 5
    start_period: 10s
  volumes:
    - asset-service-db-data:/var/lib/postgresql/data
  networks:
    - internal-network

```

Architektura systemu zakłada wzorzec *Database Per Service*, opisany w podrozdziale 5.5, dlatego każdemu serwisowi towarzyszy odrębna instancja bazy PostgreSQL. Aby uniknąć wyścigu kontenerów podczas startu środowiska, zależności między kontenerami wyrażono za pomocą dyrektywy `depends_on` z warunkami. Mikroserwis uruchamiany jest dopiero po osiągnięciu przez jego bazę oczekiwanego, predefiniowanego stanu (`condition: service_healthy`), weryfikowanego mechanizmem typu `healthcheck` wykonującym wbudowane polecenie Postgres `pg_isready`. Trwałość danych w kontenerach bazodanowych zapewniają tzw. określone/nazwane wolumeny (ang. *named volumes*), odporne na ponowne tworzenie kontenerów, natomiast parametr `restart: always` gwarantuje automatyczne wznowienie usługi po awarii lub restarcie hosta. Silnik bazy danych dostrojono przez przekazanie zestawu parametrów konfiguracyjnych (np. `shared_buffers`, `effective_cache_size`), dostosowanych do zasobów serwera produkcyjnego. Warto zwrócić uwagę na konsekwentne przypinanie wersji obrazów bazowych (np. `postgres:15.17-alpine`), które zwiększa powtarzalność wdrożeń i eliminuje niejawne zmiany, które mogłyby pojawić się bez definicji konkretnej wersji.

6.5.3 Segmentacja wirtualnej sieci kontenerowej

Docelowo, ruch sieciowy między kontenerami poddano segmentacji, izolując poszczególne grupy usług w odrębnych sieciach Docker. Listing 53 przedstawia definicję wszystkich sieci wykorzystanych w środowisku produkcyjnym.

Listing 53 Segmenty wirtualnej sieci kontenerowej w środowisku produkcyjnym. Źródło: Opracowanie własne.

```

networks:
  observability:
    name: observability
  backend:
    name: backend_network
  frontend:
    name: frontend_network

```

```

internal-network:
  name: internal_network
  internal: true
s3-internal:
  name: s3-internal
  internal: true
traefik-socket:
  name: traefik_socket
  internal: true

```

Kluczowe znaczenie ma flaga `internal: true`, która powoduje utworzenie sieci Docker pozbawionej domyślnej łączności zewnętrznej. Kontenery podłączone wyłącznie do takiej sieci mogą komunikować się z innymi kontenerami w tym samym segmencie, lecz nie posiadają domyślnej trasy wyjściowej poza tę sieć, tj. trasy `0.0.0.0/0`. Sieć `backend_network` stanowi wspólną magistralę komunikacyjną aplikacji, do której podłączona jest brama Traefik oraz mikroserwisy. Sieć `internal_network` grupuje bazy danych, które nie publikują portów na zewnątrz i są osiągalne jedynie dla kontenerów jawnie podłączonych do tego samego segmentu sieciowego, przede wszystkim dla odpowiednich mikroserwisów. Analogicznie, sieć `s3-internal` izoluje wewnętrzne komponenty magazynu plików S3 SeaweedFS, spośród których jedynie usługa bramy S3 została dodatkowo podłączona do magistrali `backend_network`, pełniąc rolę kontrolowanego punktu styku. Sieć `observability` obsługuje przepływ danych telemetrycznych do kolektora OpenTelemetry, natomiast sieć `traefik_socket` została wydzielona wyłącznie na potrzeby komunikacji bramy z usługą pośredniczącą w dostępie do demona Dockera. Takie rozdzielanie ogranicza skutki ewentualnego przejścia pojedynczego kontenera, ponieważ nie daje ono automatycznie swobodnego dostępu do pozostałych segmentów systemu, w szczególności do warstwy bazodanowej.

6.5.4 Potok CI/CD

Procesy budowania i wdrażania systemu zostały w pełni zautomatyzowane przy pomocy mechanizmu GitHub Actions, realizując koncepcję CI/CD opisaną w podrozdziale 4.13. Proces podzielono na dwa współpracujące przepływy (ang. *workflows*). Pierwszy z nich, wyzwalany przy każdym zaakceptowanym PR do gałęzi głównej `main` i odpowiada za budowę oraz publikację obrazów kontenerowych wszystkich mikroserwisów. Listing 54 przedstawia jego skróconą definicję.

Listing 54 Skrócony przepływ budowy i publikacji obrazów kontenerowych w Github Container Registry (GHCR). Źródło: Opracowanie własne.

```

on:
  push:
    branches: [ "main" ]

jobs:
  build-and-push:
    runs-on: ubuntu-24.04-arm
    permissions:
      contents: read
      packages: write
    strategy:
      fail-fast: false
    matrix:
      include:
        - service: asset-service
          dockerfile: ./asset-service/Dockerfile
        - service: frontend-service
          dockerfile: ./frontend-service/Dockerfile.prod
    # [...] pozostałe mikroserwisy

```

```

steps:
  - uses: actions/checkout@v5
  - name: Logowanie do rejestru obrazów
    uses: docker/login-action@v3
    with:
      registry: ghcr.io
      username: ${ github.actor }
      password: ${ secrets.GITHUB_TOKEN }
  - uses: docker/setup-buildx-action@v3
  - name: Budowa i publikacja obrazu
    uses: docker/build-push-action@v5
    with:
      file: ${ matrix.dockerfile }
      push: true
      platforms: linux/arm64
      tags: ${ steps.meta.outputs.tags }
      cache-from: type=gha
      cache-to: type=gha,mode=max

```

Przepływ ten buduje mikroserwisy równolegle, na podstawie jednej, wspólnej definicji kroków. Zbudowane obrazy publikowane są do rejestru GitHub Container Registry i opatrywane dwoma znacznikami: ruchomym `latest` oraz niezmiennym, odpowiadającym pełnemu identyfikatorowi wersji (ang. *commit SHA*), co umożliwia jednoznaczne powiązanie wdrożonego obrazu z konkretną wersją kodu. Budowanie realizowane jest przy użyciu narzędzia Docker Buildx na maszynie o architekturze ARM, dla platformy docelowej `linux/arm64`, zgodnej z architekturą serwera produkcyjnego. Zastosowano również współdzieloną pamięć podręczną warstw (`cache-from` oraz `cache-to` typu `gha`), co skraca czas kolejnych budowań.

Drugi przepływ odpowiada za wdrożenie środowiska produkcyjnego i wyzwalany jest zdarzeniem `workflow_run` – automatycznie, po pomyślnym zakończeniu przepływie budowania obrazów. Listing 55 przedstawia jego skróconą definicję.

Listing 55 Skrócony przepływ wdrożeniowy środowiska produkcyjnego. Źródło: Opracowanie własne.

```

name: prod_deploy_job

on:
  workflow_run:
    workflows: [ "docker_image_cicd_pipeline_job" ]
    types:
      - completed
    branches:
      - main

jobs:
  deploy:
    if: ${ github.event.workflow_run.conclusion == 'success' }
    runs-on: ubuntu-latest
    steps:
      - name: "Checkout repository"
        uses: actions/checkout@v4

      - name: "Deploy docker compose file"
        uses: appleboy/scp-action@v1.0.0
        with:
          host: ${ secrets.HOST }
          username: ${ secrets.USER }

```

```

    key: ${ secrets.PROD_SERVER_SSH_KEY }}
    source: "docker-compose.prod.yaml,[..] "
    target: "/opt/app/"

- name: "Launch docker compose on host"
  uses: appleboy/ssh-action@v1.2.4
  with:
    host: ${ secrets.HOST }}
    username: ${ secrets.USER }}
    key: ${ secrets.PROD_SERVER_SSH_KEY }}
    script: |
      set -e

      cd /opt/app/

      mv docker-compose.prod.yaml docker-compose.yaml
      mv keycloak/prod/glowing-waffle-ams-realm.json [..]
      echo "ACME_EMAIL=${ secrets.ACME_EMAIL }}" > .env
      echo "RABBITMQ_FQDN=${ secrets.RABBITMQ_FQDN }}" >> .env

      [..]
      echo "REPORTING_APP_DB_PASSWORD=${ secrets.REPORTING_APP_DB_PASSWORD
}}}" >> .env
      echo "MAIL_SMTP_HOST=${ secrets.MAIL_SMTP_HOST }}" >> .env

      [..]

      envsubst < monitoring/prometheus/prometheus.yaml.prod >
/opt/app/prometheus.yaml

      [..]

      echo "${ secrets.GHCR_PAT }}" | docker login ghcr.io -u patkaniec22
--password-stdin

      docker compose pull
      docker compose up -d--remove-orphans
      docker image prune -f
      docker logout ghcr.io

```

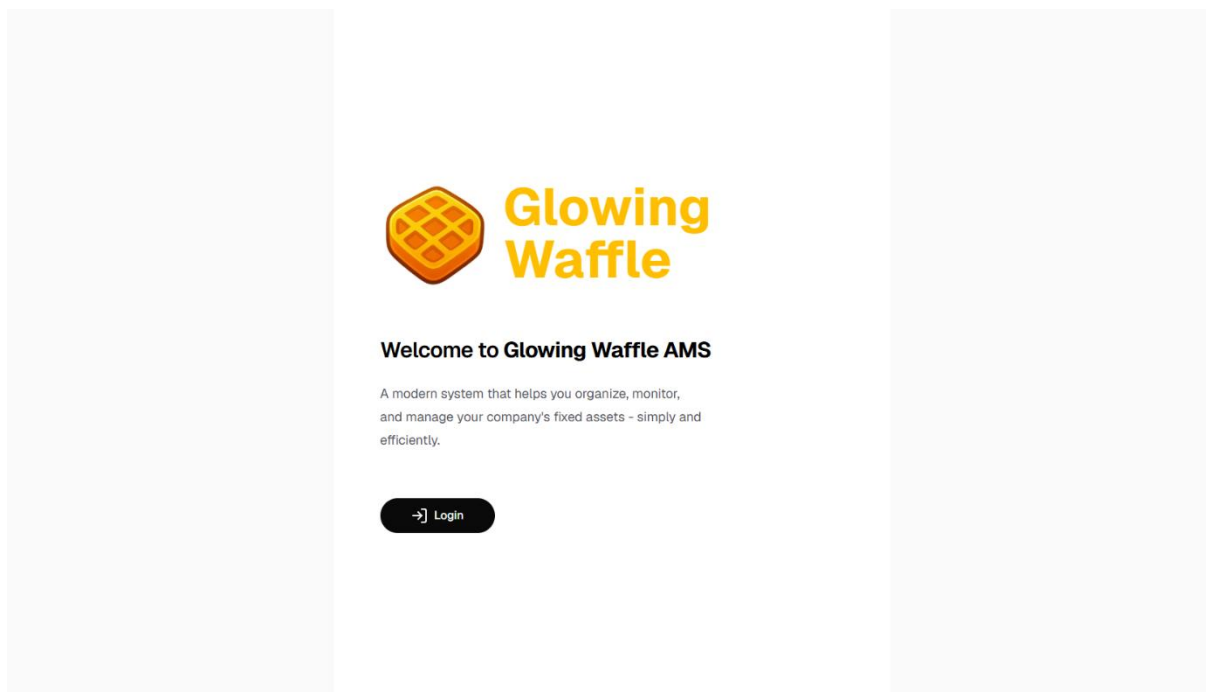
Skrypt wdrożeniowy kopiuje na serwer produkcyjny plik `docker-compose.prod.yaml` wraz z plikami konfiguracyjnymi, a następnie poprzez SSH podstawia wartości wrażliwe z zaszyfrowanych sekretów GitHub do pliku `.env`, *renderuje* szablony konfiguracji narzędziem `envsubst` i poleceniami `docker compose pull` oraz `docker compose up -d` aktualizuje działające usługi. Dzięki temu cała droga od scalonej zmiany w kodzie, przez budowę obrazów, aż po wdrożenie pozostaje w pełni zautomatyzowana i powtarzalna.

7. Interfejs użytkownika

Rozdział przedstawia widoki aplikacji webowej systemu Glowing Waffle AMS. Interfejs zaprojektowany został z myślą o codziennej pracy operacyjnej, której priorytetem była czytelność prezentowanych danych, minimalna liczba kroków potrzebna do wykonania typowej operacji oraz spójna nawigacja między obszarami systemu. Ze względu na model ról opisany w podrozdziale 3.4, wygląd poszczególnych widoków różni się w zależności od uprawnień zalogowanego użytkownika – tam, gdzie różnice są istotne, zostały one wyraźnie zaznaczone.

7.1. Logowanie i strona główna

Dostęp do systemu Glowing Waffle AMS poprzedzony jest ekranem powitalnym aplikacji, z którego użytkownik inicjuje proces uwierzytelnienia. Strona powitalna prezentuje nazwę i logotyp systemu wraz z krótkim opisem jego przeznaczenia, a jedyną dostępną na niej akcją jest przycisk *Login*, który przekierowuje użytkownika na stronę logowania udostępnianą przez serwer Keycloak. Rysunek 41 przedstawia ekran powitalny systemu.



Rysunek 41 Ekran powitalny systemu Glowing Waffle AMS. Źródło: Opracowanie własne.

Samo uwierzytelnienie odbywa się w całości po stronie Keycloak, gdzie użytkownik wprowadza dane logowania do formularza dostarczonego przez serwer autoryzacji, nie przez aplikację frontendową. Szczegółowy opis przepływu autoryzacyjnego *Authorization Code Flow* z rozszerzeniem PKCE, realizowanego w tym kroku, zawarty jest w podrozdziale 5.4. Użytkownicy po pomyślnym uwierzytelnieniu są przekierowywani na ścieżkę domyślną przypisaną do ich roli o najwyższym priorytecie – w przypadku posiadania wielu ról decyduje kolejność priorytetu zdefiniowana w systemie. Wyłącznie role administratora systemu (*ams_admin*) i zarządcy środków (*asset_manager*) lądują na pulpicie nawigacyjnym */home*. Pozostałe role trafiają bezpośrednio do obszaru odpowiadającego ich

podstawowemu zadaniu, co skraca czas dotarcia do najczęściej używanych widoków. Tabela 16 przedstawia domyślne ścieżki przekierowania dla każdej roli.

Tabela 16 Domyślne ścieżki przekierowania po zalogowaniu dla poszczególnych ról. Źródło: Opracowanie własne.

Rola	Domyślna ścieżka po zalogowaniu
ams_admin	/home
asset_manager	/home
inventory_auditor	/campaigns-flow
loan_approver	/loans-flow
incident_handler	/incidents-flow
Inventory_counter	/personal/my-campaigns
ams_user	/personal/my-assets

Strona główna pełni rolę pulpitu nawigacyjnego (ang. *dashboard*) agregującego kluczowe informacje o stanie systemu. Jej zawartość jest generowana dynamicznie na podstawie ról zalogowanego użytkownika – dane pobierane są równolegle wyłącznie z tych mikroserwisów, do których zalogowany użytkownik posiada uprawnienia, co zapobiega generowaniu błędów autoryzacyjnych dla sekcji niedostępnych dla danej roli.

Pulpit składa się z trzech poziomów. Pierwszy poziom stanowi siatka kafelków statystycznych prezentujących liczniki kluczowych zasobów – łączna liczba środków trwałych, kategorii, lokalizacji, personelu, kampanii inwentaryzacyjnych oraz incydentów. Każdy kafelek jest klikany i kieruje użytkownika bezpośrednio do odpowiadającego mu widoku. Tabela 17 przedstawia przypisanie kafelków do ról uprawnionych do ich wyświetlenia.

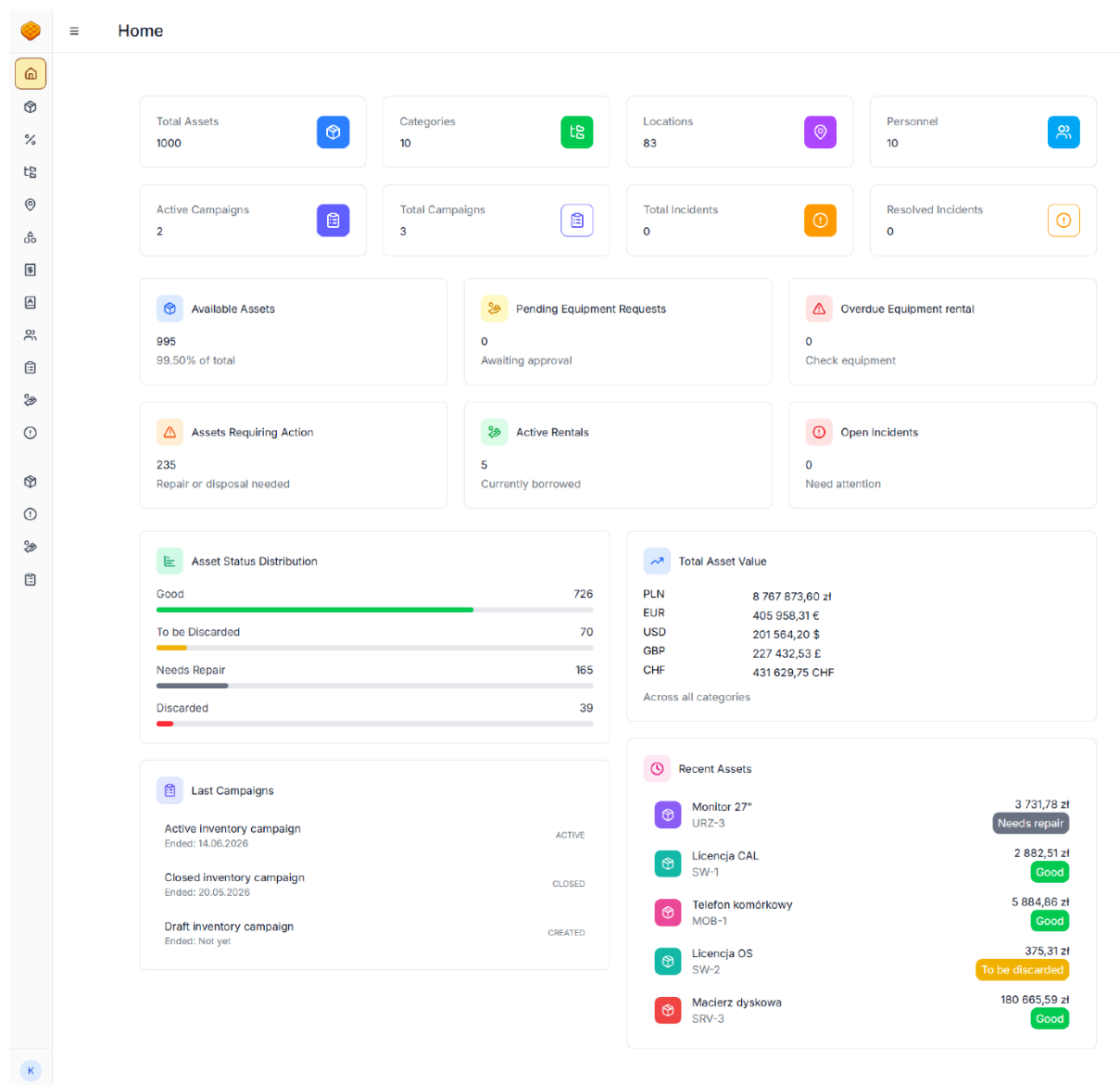
Tabela 17 Kafelki statystyczne pulpitu nawigacyjnego i uprawnione do nich role. Źródło: Opracowanie własne.

Kafelek	Uprawnione role
Total Assets, Categories, Locations	ams_admin, asset_manager, inventory_auditor, loan_approver, incident_handler
Personnel	ams_admin
Active Campaigns, Total Campaigns	ams_admin, inventory_auditor, inventory_counter
Total Incidents, Resolved Incidents	ams_admin, incidents_handler

Drugi poziom zawiera karty informacyjne widoczne dla ról operatorskich: liczbę wolnych (nieprzypisanych) środków trwałych wraz z ich procentowym udziałem w całości ewidencji, liczbę środków trwałych które posiadają status do naprawy lub konieczna utylizacja, liczbę oczekujących na rozpatrzenie wniosków o wypożyczenie, liczbę przeterminowanych i aktywnych wypożyczeń oraz liczbę otwartych incydentów.

Trzeci poziom prezentuje dwa panele analityczne po lewej stronie oraz dwa panele informacyjne po prawej. Panel rozkładu stanów technicznych środków trwałych (ang. *Asset Status Distribution*) wizualizuje za pomocą wykresu poziomego proporcje środków w stanach: dobry, do likwidacji, wymaga naprawy oraz zlikwidowany. Panel ostatnich kampanii przedstawia pięć ostatnich kampanii

inwentaryzacyjnych wraz z datą zakończenia i statusem. Po prawej stronie prezentowana jest łączna wartość ewidencji z podziałem na waluty (ang. *Total Asset Value*) oraz lista pięciu ostatnio dodanych środków trwałych z ich etykietą systemową (ang. *tag*), ceną netto i kolorem stanu technicznego. Rysunek 42 prezentuje wygląd całości strony pulpitu nawigacyjnego.



Rysunek 42 Pulpit nawigacyjny systemu Glowing Waffle AMS w widoku administratora. Źródło: Opracowanie własne.

7.2. Zarządzanie środkami trwałymi

Obszar zarządzania środkami trwałymi jest centralną częścią systemu, dostępną dla ról `ams_admin`, `asset_manager`, `inventory_auditor` oraz `loan_approver`. Widoki tego obszaru obejmują zestaw widoków konfiguracyjnych stanowiących wymagane elementy systemu, listę środków trwałych z możliwością filtrowania, wyszukiwania i przełączania trybu prezentacji, formularza dodawania oraz edycji środka oraz rozbudowany widok szczegółowy.

7.2.1 Widoki konfiguracyjne systemu ewidencji

Przed przystąpieniem do rejestracji środków trwałych administrator lub zarządca środków musi skonfigurować elementy słownikowe systemu, które są wymagane podczas tworzenia środka. Zależności potrzebne do utworzenia środka obejmują: stawkę VAT, dokument zakupu, lokalizację oraz kategorię. Nieobowiązkowo formularz środka przyjąć może także typ, powiązany z kategorią oraz wartości pól niestandardowych wprowadzane podczas uzupełniania drugiej części formularza.

Dokumenty zakupów (ang. *Purchases*) stanowią rejestry faktur zakupowych grupujące środki trwale nabyte w ramach jednej transakcji. Widok listy zakupów oferuje trzy tryby prezentacji – tabelaryczny, kafelkowy i listowy – przełączane w nagłówku sekcji i zapamiętywane przez przeglądarkę. Formularz tworzenia i edycji zakupu zawiera pola numeru faktury, daty, waluty, wartości netto i brutto oraz opcjonalnego opisu i przepisanych środków trwałych. Rysunek 43 przedstawia wygląd formularzy w wersji dla tworzenia i edycji zakupu.

Rysunek 43 przedstawia dwa formularze: 'Add Purchase' (po lewej) i 'Edit Purchase' (po prawej), oddzielone pionową linią. Formularz 'Add Purchase' zawiera pola: 'Invoice Number *' (pusty), 'Purchase Date *' (format dd.mm.rrrr z ikonami kalendarza i zegara), 'Total Net Cost *' (pusty), 'Total Gross Cost *' (pusty), 'Currency *' (wybór PLN), 'Description' (pusty tekstowy), 'Linked Assets' (pole 'Search asset...' z przyciskiem 'Open'). Formularz 'Edit Purchase' zawiera pola: 'Invoice Number *' (wartość FV/0001/06/2023), 'Purchase Date *' (wartość 28.06.2023 z godziną 17:38), 'Total Net Cost *' (wartość 16108,76), 'Total Gross Cost *' (wartość 19813,77), 'Currency *' (wybór PLN), 'Description' (wartość Zakup od Jabra), 'Linked Assets' (wartość 6 selected z przyciskiem 'Open'). Na dole każdego formularza znajdują się przyciski 'Cancel' i 'Save'.

Rysunek 43 Formularz tworzenia i edytowania dokumentu zakupu. Źródło: Opracowanie własne.

Widok szczegółowy zakupu prezentuje dane finansowe faktury zestawione z sumą wartości przypisanych środków, z automatycznym ostrzeżeniem o rozbieżności tych kwot. Rysunek 44 przedstawia widok szczegółowy zakupu wraz z rozbieżnością i ostrzeżeniem w postaci powiadomienia typu *Toast*.

Purchases
This purchase has a discrepancy in its value. Please review the net and gross invoice differences.

[← Back to Purchases](#)

FVJ0001/06/2023

Purchase Order Details

Edit

Delete

Invoice FVJ0001/06/2023	Date 28.06.2023	Currency PLN	Assets 6
Total Net Cost 16 108,76 zł	Total Gross Cost 19 813,77 zł		
Net assigned assets sum 94 077,83 zł	Gross assigned assets sum 105 654,83 zł		
Difference in net invoice (+) 77 969,07 zł	Difference in gross invoice (+) 85 641,06 zł		
Description Zakup od Jabra			

Assets in this Purchase

Click on any asset to view its details

[Download](#)
[Import from CSV](#)
[+ Create and Add](#)
[Add Existing](#)

Search by...

Asset Tag T1	Name T1	Type T1 @	Category T1 @	Assigned To T1	Condition T1 @	Net Price T1	Gross Price T1	Currency T1 @
NAR2-19	Mierik sieciowy Tester okablowania/siec — Fluke	Mierik sieciowy	Narzędzia	-	To Be Discarded	1 835,84	1 788,71	PLN (z)
SRV-49	Micierz dyskowa System przechowywania danych — Dell	Macierz dyskowa	Serwerownia	-	Good	61 836,48	66 783,38	PLN (z)
SRV-74	UPS Zasilacz awaryjny — APC	UPS	Serwerownia	-	Good	27 967,38	31 030,89	PLN (z)
URI2-156	Klawiatura Klawiatura przewodowa/bezprzewodowa — J...	Klawiatura	Urządzenia	-	Good	1 470,78	1 568,44	PLN (z)
NAR2-43	Mierik sieciowy Tester okablowania/siec — Fluke	Mierik sieciowy	Narzędzia	-	Needs Repair	725,61	743,77	PLN (z)

Showing 6 records
Rows per page: 5 < >
Page 1 of 2 < >

Rysunek 44 Widok szczegółów zakupu z widocznym ostrzeżeniem o rozbieżności wartości oraz tabelą powiązanych środków trwałych. Źródło: Opracowanie własne.

Lokalizacje (ang. *Locations*) opisują fizyczne miejsce przechowywania środków w trójpoziomowej hierarchii: budynek, piętro i pomieszczenie. Podobnie jak dokumenty zakupów, widok lokalizacji oferuje trzy tryby prezentacji. Formularz tworzenia lokalizacji zawiera trzy pola odpowiadające hierarchii oraz opis. Widok szczegółów lokalizacji prezentuje jej dane oraz tabelę przypisanych środków z możliwością przejścia do widoku każdego z nich. Rysunek 45 listę lokalizacji zaprezentowaną w trybie listowym.

Locations
Manage physical locations of assets

Search by... Filters

Building	Floor	Room	Description
A	-	Kuchnia A001	Budynek główny, piętro 0
A	-	Open space A002	Budynek główny, piętro 0
A	-	Recepcja A003	Budynek główny, piętro 0
A	-	Magazyn A004	Budynek główny, piętro 0
A	-	Sala konf. A005	Budynek główny, piętro 0

Showing 83 records Rows per page 5 Page 1 of 17

Rysunek 45 Widok listy lokalizacji prezentowany w formie listowym. Źródło: Opracowanie własne.

Kategorie (ang. *Categories*), typy zasobów (ang. *Asset Types*) oraz definicje pól (ang. *Field Definitions*) to elementy ściśle ze sobą powiązane – kategoria grupuje środki i definiuje przedrostek etykiety (ang. *tag prefix*), typ jest opcjonalnym uszczegółowieniem kategorii nadpisującym jej przedrostek, a definicja pola jest szablonem niestandardowego atrybutu przypisywanym do kategorii lub typu. Wszystkie trzy widoki listy prezentowane są w trybie kafelkowym, bądź tabelarycznym, który dla kategorii eksponuje kolor wizualny stanowiący kluczowy identyfikator w całym interfejsie. Widoki szczegółów każdego z tych elementów prezentują jego dane oraz tabele powiązanych elementów – kategoria pokazuje przypisane środki i typy, typ pokazuje przypisane definicje pól, definicja pokazuje typy i kategorie korzystające z niej. Rysunek 46 przedstawia widok szczegółowy pól definicji jako reprezentanta ilustrującego wygląd powyżej wymienionych trzech elementów – kategorii, typów zasobów oraz definicji pól.

Definitions

← Back to Definitions

Numer seryjny
a384178f-bc1b-4935-980b-d7835080079c

Edit
 Delete

Definition Information

Name	Label
Numer seryjny	S/N
Type	
STRING	

Assigned Types (1)

Columns

Name	Description	Assigned Definitions
Laptop	Komputer przenośny	2

Showing 1 records
Rows per page 5
Page 1 of 1

Assigned Categories (4)

Columns

Name	Description	Color
Sprzęt IT	Komputery, serwery, urządzenia sieciowe	
Urządzenia	Monitory, klawiatury, myszy, słuchawki	
Network gear	Switche, routery, access pointy, firewalle	
Serwerownia	Serwery, macierze, UPS-y, szafy rack	

Showing 4 records
Rows per page 5
Page 1 of 1

Rysunek 46 Widok szczegółowy definicji pól przedstawiający tabele powiązanych kategorii oraz typów.
Źródło: Opracowanie własne.

Stawki VAT (ang. *VAT Rate*) to najprostszy element słownikowy – każda stawka posiada wyłącznie nazwę i wartość procentową. Widok listy prezentowany jest w trybie tabelarycznym lub kafelkowym, a widok szczegółów pokazuje dane stawki oraz przypisane do niej środki trwałe, wyglądający analogicznie do widoku przedstawionego na Rysunek 46, natomiast widok kafelkowy został zaprezentowany na Rysunek 47.

VAT Rate	Name	Status
0.23 %	Stawka podstawowa	Active
0.08 %	Stawka obniżona	Active
0.05 %	Stawka obniżona II	Active
0 %	Stawka zerowa	Active
0 %	Zwolniony z VAT	Active

Rysunek 47 Widok listy stawek VAT prezentowany w formie kafelkowej. Źródło: Opracowanie własne.

7.2.2 Lista środków trwałych

Widok listy środków trwałych prezentuje ewidencję w formie tabeli z możliwością sortowania każdej kolumny, co zostało zilustrowane na Rysunku 48. Dostępne kolumny obejmują etykietę systemową środka, nazwę, stan techniczny, datę ostatniej kampanii inwentaryzacyjnej, kategorię, lokalizację, typ, cenę netto, walutę, stawkę VAT oraz status przypisania. Kolumny – stan techniczny, data ostatniej kampanii, typ, kategoria, cena netto i brutto oraz waluta – wyposażone są w dedykowane filtry kolumnowe. Kolumny stanu technicznego, typu, kategorii i waluty udostępniają filtr wielokrotnego wyboru (ang. *multi filter*), kolumna daty ostatniej kampanii – filtr zakresu dat (ang. *date range filter*), natomiast kolumny cen brutto i netto – filtr wyboru zakresu numerycznego (ang. *number range filter*). Wyszukiwanie globalne po wszystkich kolumnach jednocześnie dostępne jest przez pole tekstowe z napisem *Search by* w nagłówku tabeli.

Asset Tag	Name	Condition	Last Campaign Date	Type	Category	Unit Price Net	Unit Price Gross	Currency	Vat Rate	Assigned	Actions
UR2-2	Monitor 27"	Needs repair	-	Monitor 27"	Urządzenia	3 721,78	3 918,37	PLN (zł)	5%	Available	[Icons]
SW-1	Licencja CAL	Good	-	Licencja CAL	Software	2 862,31	3 113,11	PLN (zł)	8%	Available	[Icons]
MOB-1	Telefon komórkowy	Good	-	Telefon komórkowy	Telefony	5 884,88	6 355,65	PLN (zł)	8%	Available	[Icons]
SW-2	Licencja OS	To be discarded	-	Licencja OS	Software	375,31	405,33	PLN (zł)	8%	Available	[Icons]
SRV-3	Macierz dyskowa	Good	-	Macierz dyskowa	Sprzęt serwerowy	180 665,59	180 665,59	PLN (zł)	0%	Available	[Icons]

Rysunek 48 Widok listy środków trwałych w trybie tabeli z widocznym paskiem wyszukiwania globalnego, nagłówkami kolumn z ikonami sortowania oraz przyciskami akcji w ostatniej kolumnie. Źródło: Opracowanie własne.

Widok listy oferuje trzy tryby prezentacji danych, przełączane przyciskami w nagłówku sekcji: tabelaryczny (ang. *table*), kafelkowy (ang. *tiles*) oraz listowy (ang. *list*). Wybrany tryb jest zapamiętywany przez przeglądarkę i przywracany przy kolejnych odwiedzinach widoku. Tryb tabelaryczny prezentuje tabelę przedstawiającą środki w postaci wierszy z kolorowym oznaczeniem kategorii i stanu technicznego oraz podstawowymi danymi, natomiast tryb listowy wyświetla środki w poziomych wierszach z rozszerzonymi metadanymi – lokalizacją, typem i datą ostatniej kampanii, natomiast kafelki wyświetlają elementy w siatce prezentującej najważniejsze elementy i przyciskiem rozwinięcia zawierającego informacje o typie oraz ostatniej kampanii.

7.2.3 Dodawanie i edycja środków trwałych

Tworzenie i edycja środka trwałego realizowane są przez dwuetapowy formularz modalny. Pierwszy etap obejmuje pola podstawowe – nazwę, opis, kategorię, typ, lokalizację, przypisanego użytkownika, stan techniczny, powiązany zakup oraz dane finansowe. Po zapisaniu pierwszego etapu otwierany jest automatycznie drugi formularz, przeznaczony do wypełnienia pól niestandardowych przypisanych do wybranej kategorii i typu środka. Zestaw tych pól jest generowany dynamicznie na podstawie definicji skonfigurowanych przez administratora. Podział na dwa etapy pozwala na wyraźne oddzielenie danych strukturalnych środka od jego elastycznych atrybutów opisowych. Rysunek 49 przedstawia oba formularze, po lewej formularz pierwszego etapu zawierający główne informacje, po prawej formularz zawierający pola niestandardowe.

Add Asset

Name *

Description

Condition *

Good

Unit Net Price *

VAT Rate *

Search vat rate...
Open

Purchase *

Search purchase...
Open

Location *

Search location...
Open

Category *

Search category...
Open

Type

Select category before selecting type

Cancel
Save 1/2

Custom fields

Numer seryjny

RAM [GB]

Gwarancja do

dd.mm.rrrr

Cancel
Save 2/2

Rysunek 49 Dwuetapowy formularz dodawania środka trwałego: pierwszy modal z polami podstawowymi z przyciskiem "Save 1/2" oraz drugi modal z polami niestandardowymi z przyciskiem "Save 2/2". Źródło: Opracowanie własne.

7.2.4 Widok szczegółów środków trwałych

Widok szczegółów środka trwałego jest najbardziej rozbudowanym widokiem w systemie, a jego zawartość jest częściowo uzależniona od roli zalogowanego użytkownika. Widok podzielony jest na kilka sekcji.

Sekcja *Asset Information* prezentuje pełne dane ewidencyjne środka – etykietę systemową, nazwę, opis, stan techniczny, kategorię, wartość brutto i netto, stawkę VAT oraz typ. Po prawej stronie wyświetlane są karty lokalizacji i przypisanego użytkownika, a poniżej - wartości pól niestandardowych przypisanych do danego środka. Edycja danych środka oraz jego usunięcie dostępne są wyłącznie dla ról `ams_admin` i `asset_manager`. Rysunek 50 przedstawia pełną górną część widoku szczegółowego przedstawiającą wyżej opisane sekcje.

Assets

← Back to Assets

URZ-3
Monitor 29"

EditDelete

Asset Information

Asset Tag
URZ-3

Category
Urządzenia

Condition
Needs repair

Unit Gross Price
3 918,37 zł

Description
Monitor 29 cali — Dell

Asset Name
Monitor 29"

Asset Type
Monitor 27"

Unit Net Price
3 731,78 zł

VAT Rate
5%

Location

B
0
Room Open space B005
Navigate to location

Assigned To

Available
Not assigned

Custom Information

S/N (Numer seryjny)
SN-MON-90952459

Gwarancja (Gwarancja do)
2026-03-27

Rysunek 50 Górna sekcja widoku szczegółowego środka trwałego z widocznymi sekcjami Asset Information, Location, Assigned To oraz Custom Information. Źródło: Opracowanie własne.

Sekcja *Identifiers* prezentuje dwa rodzaje identyfikatorów zewnętrznych środka – kod QR oraz kod kreskowy, oba wygenerowane na podstawie UUID środka. Każdy z nich można wydrukować przy użyciu dedykowanego przycisku otwierającego okno drukowania przeglądarki, tak jak przedstawiono na Rysunek 51.

Identifiers

Print QR

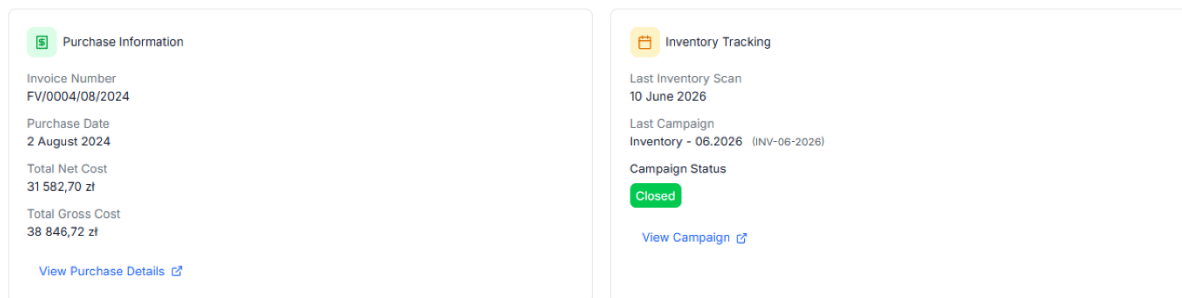

37ff042a-9aab-4f68-84c1-4f9bd949e27e

Print Barcode


N_8EKpqrT2iEwU-b2Unifg

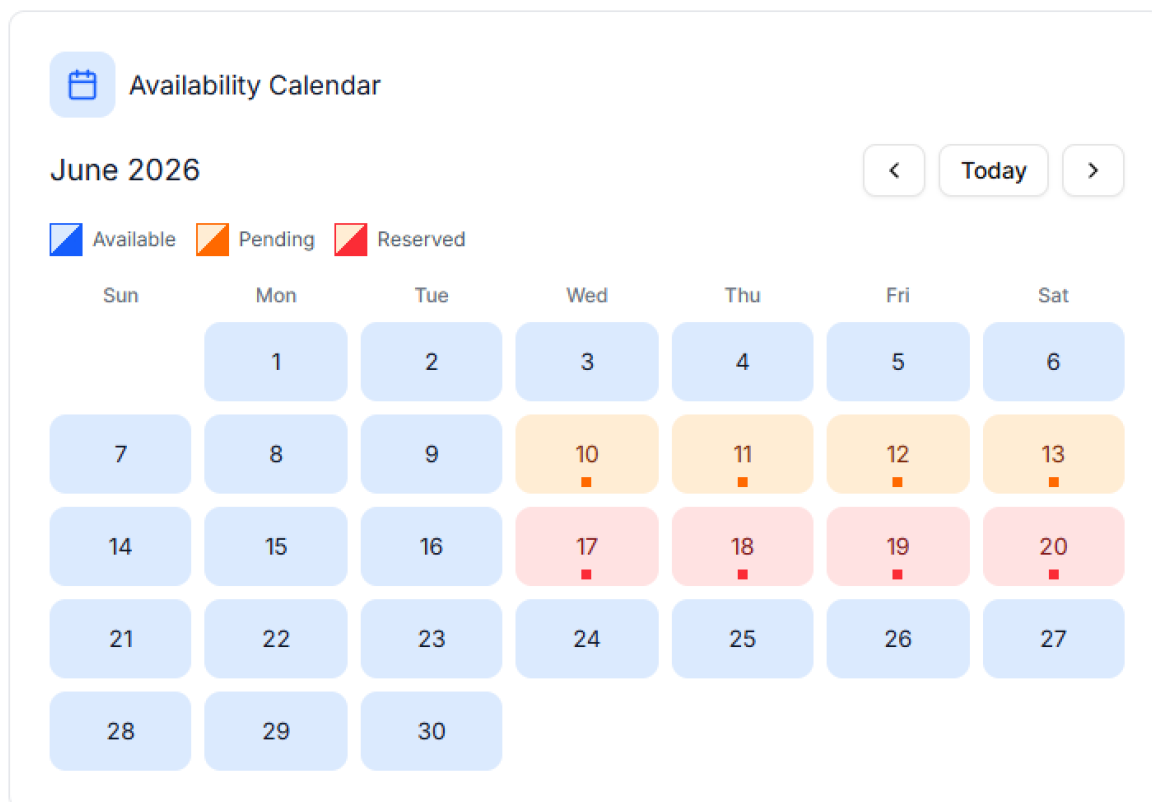
Rysunek 51 Sekcja Identifiers widoku szczegółów środka trwałego z widocznym kodem QR, kodem kreskowym oraz przyciskami drukowania. Źródło: Opracowanie własne.

Sekcja *Purchase Information* zawiera dane zakupowe środka – numer faktury, datę zakupu, cenę netto i brutto z uwzględnieniem waluty i stawki VAT. Sekcja *Inventory Tracking* prezentuje datę ostatniej kampanii inwentaryzacyjnej oraz – widoczne wyłącznie dla ról `ams_admin` i `asset_manager` – informację o aktualnie trwającej kampanii z możliwością bezpośredniego przejścia do jej widoku. Rysunek 52 przedstawia obie sekcje jedna obok drugiej.



Rysunek 52 Sekcja Purchase Information oraz Inventory Tracking widoku szczegółów środka trwałego z widocznymi informacjami o zamówieniu oraz ostatnio wykonanej kampanii inwentaryzacyjnej. Źródło: Opracowanie własne.

Sekcja *Availability Calendar* prezentuje kalendarz dostępności środka w zakresie sześciu lat (trzy lata wstecz i trzy lata naprzód), z zaznaczonymi okresami wypożyczenia. Pozwala to na szybką ocenę historii oraz planowanie przyszłych wypożyczeń. Rysunek 53 przedstawia kalendarz z zaznaczonymi okresami wypożyczeń w dwóch kolorach reprezentujących zarezerwowane wypożyczenie oraz wypożyczenie oczekujące potwierdzenia.



Rysunek 53 Sekcja Availability Calendar widoku szczegółów środka trwałego z zaznaczonymi okresami wypożyczenia na tle siatki miesięcy. Źródło: Opracowanie własne.

Sekcja *Asset History* prezentuje dziennik zdarzeń środka – chronologiczny rejestr wszystkich operacji wykonanych na środku, takich jak zmiany stanu, kampanie inwentaryzacyjne i incydenty. Sekcja dostępna jest wyłącznie dla ról `ams_admin` i `asset_manager`. Rysunek 54 ilustruje te sekcje.

Asset History					
Search by... Filters Columns					
Occurred At	Recorded At	Action	Source	Actor	Details
2026-06-10 09:07:41	2026-06-10 09:07:41	Asset updated	Asset service	Kacper Barański	Numer seryjny: SN-MGN-90949799 → SN-MGN-90952459
2026-06-10 09:07:30	2026-06-10 09:07:30	Asset updated	Asset service	Kacper Barański	Name: Monitor 27 → Monitor 29 Description: Monitor 27 cali — microsoft → Monitor 29 cali — dell
Showing 2 records Rows per page 5 Page 1 of 1 << < > >>					

Rysunek 54 Sekcja Asset History widoku szczegółów środka trwałego z przykładowymi oznaczeniami zmian. Źródło: Opracowanie własne.

Sekcja *Request History*, przedstawiona na Rysunku 55, widoczna dla ról `ams_admin`, `asset_manager` i `loan_approver`, prezentuje historię wniosków o wypożyczenie powiązanych z danym środkiem wraz ze statusami i datami.

Request History				
Search by... Filters Columns				
Occurred At	Event Type	Actor	Equipment Request	Finalized Request
2026-06-10 09:09:32	Rejected	Kacper Barański	2026-06-10 → 2026-06-13 66943516-9edd-4a56-9ab0-ec237c1181f4	-
2026-06-10 09:09:12	Returned	Kacper Barański	2026-06-17 → 2026-06-20 66a6f04e-10d1-4399-b6f5-5401032c2056	2026-06-17 → 2026-06-20 c50e2ff3-dea6-4b95-ba6f-9fa604cfea19
2026-06-10 09:08:39	Issued	Kacper Barański	2026-06-17 → 2026-06-20 66a6f04e-10d1-4399-b6f5-5401032c2056	2026-06-17 → 2026-06-20 c50e2ff3-dea6-4b95-ba6f-9fa604cfea19
2026-06-10 09:06:42	Approved	Kacper Barański	2026-06-17 → 2026-06-20 66a6f04e-10d1-4399-b6f5-5401032c2056	-
2026-06-10 09:06:30	Requested	Kacper Barański	2026-06-17 → 2026-06-20 66a6f04e-10d1-4399-b6f5-5401032c2056	-
2026-06-10 09:06:15	Requested	Kacper Barański	2026-06-10 → 2026-06-13 66943516-9edd-4a56-9ab0-ec237c1181f4	-
Showing 6 records Rows per page 10 Page 1 of 1 << < > >>				

Rysunek 55 Sekcja Request History widoku szczegółów środka trwałego z wypisanymi akcjami związanymi wypożyczeniami. Źródło: Opracowanie własne.

7.3. Kampanie inwentaryzacyjne

Obszar kampanii inwentaryzacyjnych dostępny jest dla ról `ams_admin` oraz `inventory_auditor`, `inventory_counter`, przy czym zakres widocznych akcji różni się istotnie w zależności od roli. Audytor i administrator zarządzają całym cyklem życia kampanii, natomiast inwentaryzator uczestniczy wyłącznie w czynnym etapie skanowania.

7.3.1 Lista kampanii

Widok listy kampanii prezentowany jest wyłącznie w trybie tabelarycznym, co odróżnia go od większości innych widoków systemu. Każdy wiersz listy prezentuje nazwę kampanii, jej status, datę

rozpoczęcia, liczbę przypisanych środków i lokalizacji oraz przypisanych uczestników, a widok ten możemy zaobserwować na Rysunek 56.

Name	Code	Status	Start Date	End Date	Created By	Actions
Kampania 1	CAMP/2026/04/1	Closed	2026-04-24	2026-04-23	Tomasz Gajewski (s27751@pjwstk.edu.pl)	[Icon]
Test	Test	Active	2026-05-17	-	Kacper Barański (s28166@pejot.edu.pl)	[Icon]
Kampania 1	CAMP/05/1	Active	2026-05-29	-	Tomasz Gajewski (s27751@pjwstk.edu.pl)	[Icon]

Rysunek 56 Widok listy kampanii inwentaryzacyjnych w trybie tabelarycznym z widocznym statusem każdej kampanii, datą rozpoczęcia i zakończenia oraz osobą tworzącą. Źródło: Opracowanie własne.

Formularz tworzenia i edycji kampanii zawiera pola: nazwa, data rozpoczęcia, wielokrotny wybór środków trwałych, wielokrotny wybór lokalizacji objętych kampanią oraz wielokrotny wybór uczestników spośród użytkowników systemu.

7.3.2 Cykl życia kampanii i widok szczegółów

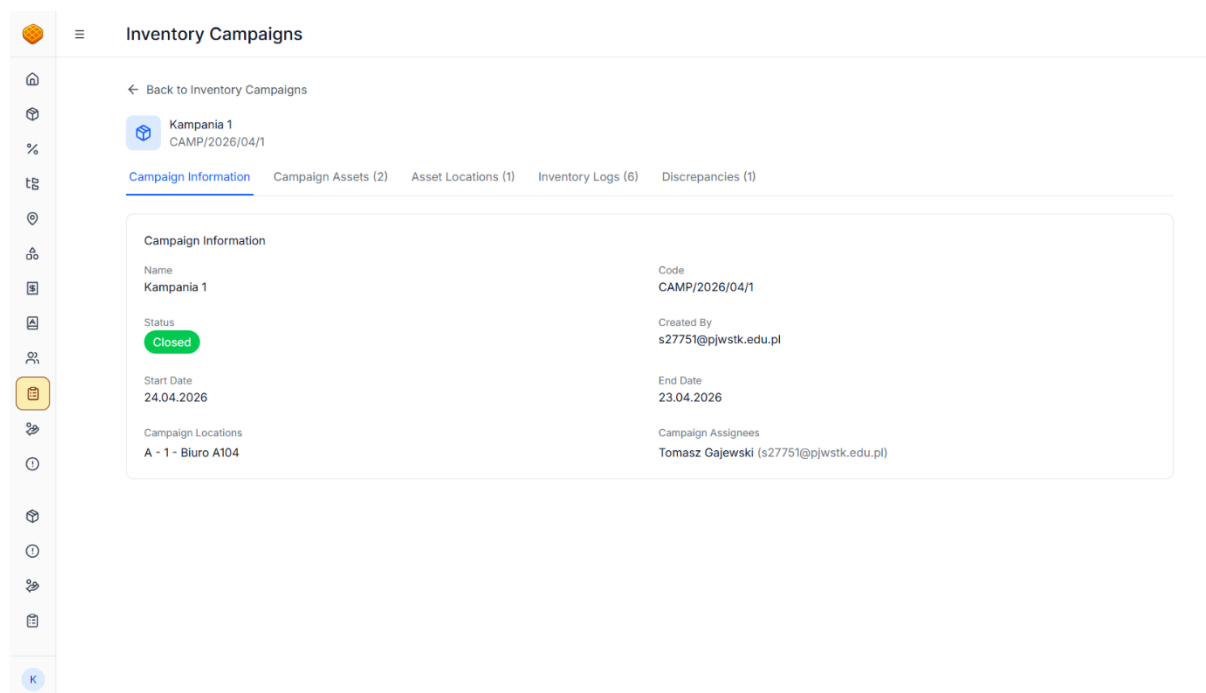
Kampania przechodzi przez cztery statusy: CREATED, ACTIVE, CALCULATING_RESULTS oraz CLOSED. Dostępne akcje w nagłówku widoku szczegółów są uzależnione od aktualnego statusu, co przedstawia Tabela 18.

Tabela 18 Akcje dostępne w widoku szczegółów kampanii w zależności od statusu. Źródło: Opracowanie własne.

Status	Dostępne akcje
CREATED	Edycja kampanii, Start kampanii
ACTIVE	Stop kampanii, Skan środków trwałych
CALCULATING_RESULTS	Synchronizacja wyników
CLOSED	Podgląd wyników

Widok szczegółów kampanii podzielony jest na pięć zakładek, co ilustruje Rysunek 57. Zakładka *Campaign Information* prezentuje podstawowe dane kampanii - nazwę, status, daty, przypisane lokalizacje oraz uczestników. Zakładka *Campaign Assets* wyświetla tabelę środków objętych kampanią z oznaczeniem statusu inwentaryzacji każdego z nich – środki już zeskanowane oznaczone są wizualnie inaczej niż te oczekujące na weryfikację. Zakładka *Asset Locations* prezentuje przypisane do kampanii lokalizacje wraz z postępem skanowania w każdej z nich. Zakładka *Inventory Logs* zawiera chronologiczny dziennik wszystkich operacji skanowania wykonanych w trakcie kampanii – kto

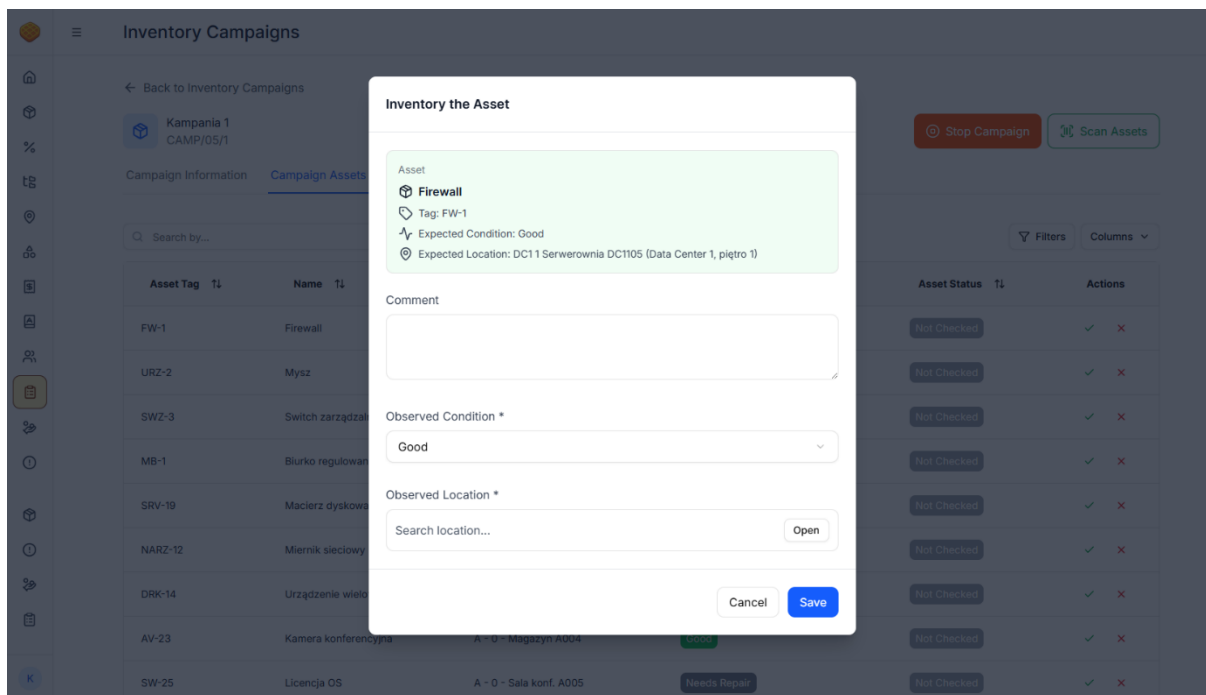
zeskanował dany środek, kiedy i z jakiego miejsca. Zakładka *Discrepancies* prezentuje tabelę rozbieżności wykrytych po zakończeniu kampanii – środków, które nie zostały odnalezione w oczekiwanej lokalizacji lub których stan różni się od zapisanego w ewidencji.



Rysunek 57 Widok szczegółów kampanii z widocznym paskiem zakładek, nagłówkiem z przyciskami akcji zależnymi od statusu oraz zakładką Campaign Information. Źródło: Opracowanie własne

7.3.3 Skanowanie środka trwałego w trakcie kampanii

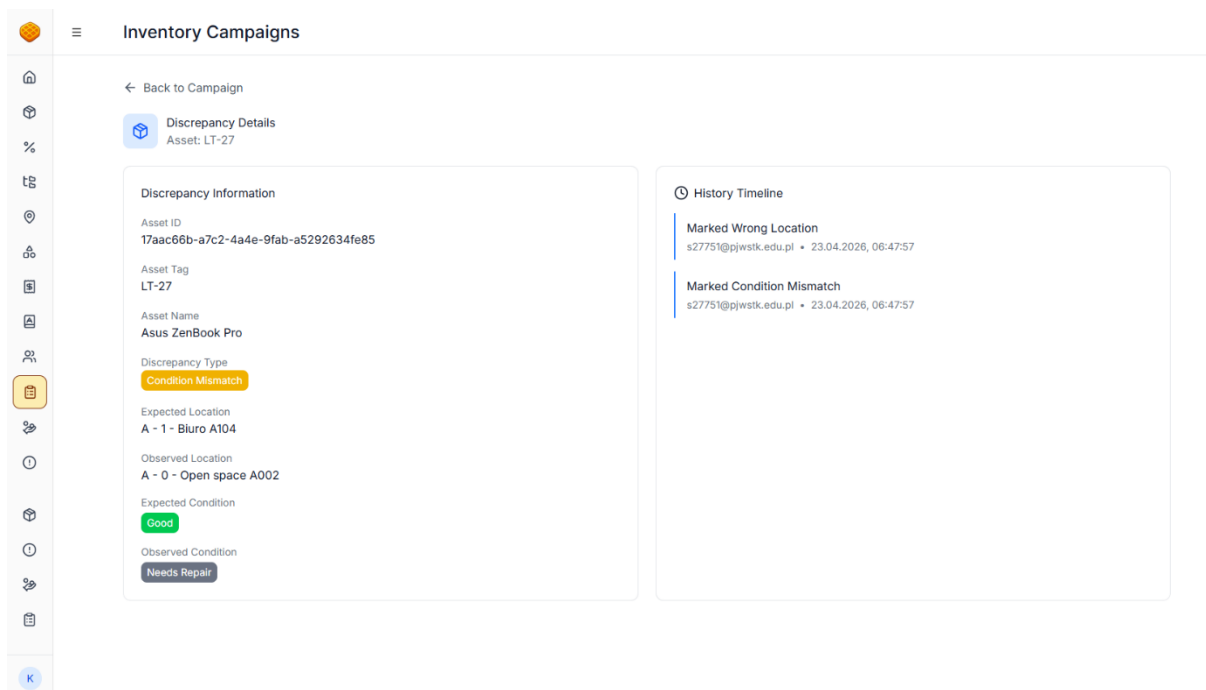
Kluczową funkcjonalnością widoku aktywnej kampanii jest możliwość skanowania środków trwałych bezpośrednio w przeglądarce, bez konieczności instalowania dedykowanej aplikacji mobilnej. Po kliknięciu przycisku *Scan Assets* otwierany jest komponent skanera, skonfigurowany w trybie kampanijnym. Po pomyślnym odczytaniu kodu QR lub kreskowego środka system weryfikuje, czy dany środek należy do bieżącej kampanii, a następnie otwiera formularz modalny potwierdzenia ze szczegółami środka – jego nazwą, etykietę systemową, oczekiwaną lokalizacją oraz stanem technicznym. Użytkownik potwierdza obecność środka lub zgłasza rozbieżność, po czym środek oznaczany jest jako zinwentaryzowany na liście zakładki *Campaign Assets*. Rysunek 58 prezentuje wygląd modala potwierdzenia skanowania środka trwałego podczas kampanii.



Rysunek 58 Modal potwierdzenia skanowania środka w trakcie aktywnej kampanii z widocznymi danymi środka, oczekiwaną lokalizacją oraz przyciskami potwierdzenia i zgłoszenia rozbieżności. Źródło: Opracowanie własne.

7.3.4 Widok szczegółów rozbieżności

Rozbieżności wykryte w trakcie i po zakończeniu kampanii dostępne są w tabeli pod dedykowaną zakładką *Discrepancies*, gdzie każda z nich posiada własny osobny podwidok, otwierany przez kliknięcie ikony oka. Prezentuje on szczegóły rozbieżności – typ (np. środek nieodnaleziony, błędna lokalizacja), oczekiwaną lokalizację środka oraz odniesienie do środka trwałego, którego dotyczy. W trakcie jednej kampanii inwentaryzacyjnej status środka trwałego może zostać zmieniony więcej niż raz, jeśli takowa zamiana wprowadza kolejną rozbieżność jej status w tym panelu zostanie zaktualizowany, a prawy panel pokazujący oś czasu zapisywać będzie wszystkie z nich. Podwidok rozbieżności z kilkoma zmianą wyświetlanymi na osi czasu zaprezentowany jest na Rysunek 59.



Rysunek 59 Podwidok szczegółów rozbieżności z widocznym typem rozbieżności, oczekiwaną lokalizacją środka oraz przyciskiem powrotu do kampanii. Źródło: Opracowanie własne.

7.4. Wypożyczenia

Obszar wypożyczeń obsługuje pełny cykl życia wniosku o wypożyczenie środka trwałego – od złożenia wniosku przez pracownika, przez decyzję operatora, transfer środka, aż po finalizację i zwrot. Widok dostępny jest przede wszystkim dla ról `ams_admin` i `loan_approver`, natomiast każdy zalogowany użytkownik z rolą `ams_user` może składać własne wnioski przez sekcję widoków osobistych, opisaną w podrozdziale 7.7.

Cały obszar wypożyczeń zorganizowany jest w czterech zakładkach prezentowanych w ramach jednego widoku: *Equipment Requests*, *Decisions*, *Transfers* oraz *Finalized Requests*. Każda zakładka wyświetla w nawiasie liczbę rekordów, co pozwala na szybką orientację w stanie systemu bez przełączania się między widokami. Rysunek 60 ilustruje widok wypożyczeni raz z dostępnymi zakładkami.

Equipment Requests Management
Manage equipment requests, decisions, and transfers

+ New Request

Equipment Requests 9 Decisions 8 Transfers 0 Finalized Requests 8

Search by...

Filters Columns

Assets	Created At	Requested By	Justification	Status	Period	Next Step	Actions
LT-5 (Laptop)	2026-04-22 22:17:54	Piotr Narkaniec (s28145@cjwsk.edu.pl)	Needed for work	Approved	2026-04-28 - 2026-04-30	Decision approved Pick up asset(s) on or after the start date	
LT-28 (Dell Inspiron 14u)	2026-04-23 04:32:25	Tomasz Gajewski (s2775@cjwsk.edu.pl)	Potrzebuję bardzo	Approved	2026-04-23 - 2026-04-30	Decision approved Pick up asset(s) on or after the start date	
LT-27 (Asus ZenBook Pro)	2026-04-23 04:37:49	Tomasz Gajewski (s2775@cjwsk.edu.pl)	Potrzebny na wyjazd służbowy	Rejected	2026-04-23 - 2026-04-30	Decision rejected No further action	
LT-27 (Asus ZenBook Pro)	2026-04-23 04:59:59	Tomasz Gajewski (s2775@cjwsk.edu.pl)	Niezbędny na wyjazd służbowy	Approved	2026-04-23 - 2026-04-30	Decision approved Pick up asset(s) on or after the start date	
LT-27 (Asus ZenBook Pro)	2026-04-23 08:44:58	Piotr Narkaniec (s28145@cjwsk.edu.pl)	Needed for work	Pending	2026-05-01 - 2026-05-04	Awaiting administrator decision	

Showing 9 records

Rows per page: 5 Page 1 of 2

Rysunek 60 Widok główny obszaru wypożyczeń z widocznym paskiem czterech zakładek, licznikami rekordów oraz tabelą wniosków o wypożyczenie. Źródło: Opracowanie własne.

7.4.1 Wnioski o wypożyczenie

Zakładka *Equipment Requests* prezentuje wszystkie złożone wnioski o wypożyczenie w formie tabeli z wyszukiwaniem globalnym. Każdy wiersz zawiera wnioskującego użytkownika, listę wnioskowanych środków, żądany okres wypożyczenia, uzasadnienie oraz aktualny status wniosku. Statusy wniosków to: PENDING (oczekujący), APPROVED (zatwierdzony) oraz REJECTED (odrzucony), oznaczone kolorami dla szybkiej identyfikacji.

Formularz tworzenia wniosku, przedstawiony na Rysunku 61, zawiera wielokrotny wybór środków trwałych z filtrowaniem wyłącznie do nieprzypisanych środków, limitem do maksymalnie 5 elementów, daty początku i końca wnioskowanego okresu oraz pole uzasadnienia.

Add Equipment Request

Requested Assets (Max: 5)

Show all Close

Search asset...

- URZ-3 Monitor 27"
- URZ-2 Mysz
- URZ-13 Klawiatura
- PC-1 Desktop PC

Rows: 5 Selected: 4 Page 1 of 1

Justification *

Employee onboarding starter pack.

Request Timeline *

June 2026

Available Pending Reserved

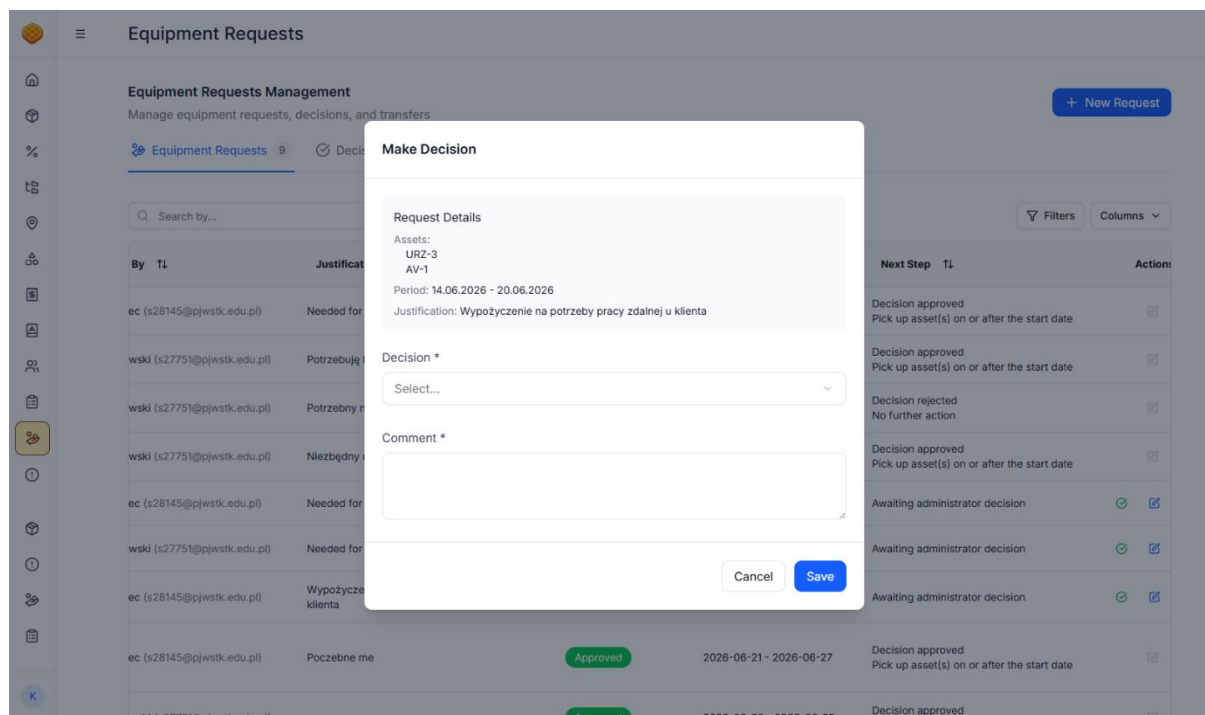
Sun	Mon	Tue	Wed	Thu	Fri	Sat
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

Selected Range: 10.06.2026 - 13.06.2026

Cancel Save

Rysunek 61 Modal tworzenia wniosku o wypożyczenie z sekcją wyboru środków trwałych, uzasadnienia i kalendarza przyjmującego okres wypożyczenia. Źródło: Opracowanie własne.

Operator zatwierdza lub odrzuca wniosek przez dedykowany modal decyzji, który w sekcji nagłówkowej prezentuje szczegóły wniosku – listę wnioskowanych środków, żądany okres oraz uzasadnienie – a poniżej udostępnia formularz decyzji z polem wyboru APPROVED/REJECTED oraz komentarzem. Rysunek 62 przedstawia modal z formularzem decyzji o wniosku.



Rysunek 62 Modal podejmowania decyzji o wniosku z sekcją szczegółów wniosku w nagłówku (lista środków, okres, uzasadnienie) oraz formularzem decyzji poniżej. Źródło: Opracowanie własne.

7.4.2 Decyzje

Zakładka *Decisions* zawiera rejestr wszystkich podjętych decyzji. Każdy wiersz tabeli prezentuje powiązany wniosek, podjętą decyzję, operatora który jej dokonał oraz datę. Zakładka pełni rolę audytową – nie udostępnia akcji modyfikacji, służąc wyłącznie do przeglądania historii decyzji.

7.4.3 Transfery

Zakładka *Transfers* obsługuje fizyczne przekazania środków trwałych między użytkownikami w ramach aktywnych wypożyczeń. Transfer rejestruje zmianę posiadacza środka bez zmiany statusu samego wypożyczenia. Formularz tworzenia transferu zawiera wybór aktywnego wypożyczenia, środka trwałego oraz docelowego użytkownika przejmującego środek.

7.4.4 Finalizowane wnioski

Zakładka *Finalized Requests*, przedstawiona na Rysunku 63, prezentuje wypożyczenia, które przeszły przez pełny cykl – od zatwierdzonego wniosku do faktycznego wydania środka. Każde wypożyczenie posiada status opisujący jego aktualny stan: SCHEDULED (zaplanowane), ACTIVE (w trakcie), OVERDUE (przeterminowane) oraz RETURNED (zwrócone). Operator może oznaczyć wypożyczenie jako wydane przyciskiem *Issue* lub jako zwrócone przyciskiem *Return*, dostępnymi bezpośrednio w wierszu tabeli. Obie akcja wymagają potwierdzenia w oknie potwierdzenia ze względu na ich nieodwracalność.

Asset	Borrower	Start Date	Due Date	Return Date	Status	Current Holder	Actions
LT-28 (Dell Inspiron 14u)	Tomasz Gajewski (s27751@pjwstk.edu.pl)	2026-04-23	2026-04-30	2026-04-23	Returned	Tomasz Gajewski (s27751@pjwstk.edu.pl)	Issue Return
LT-27 (Asus ZenBook Pro)	Tomasz Gajewski (s27751@pjwstk.edu.pl)	2026-04-23	2026-04-30	2026-04-23	Returned	Tomasz Gajewski (s27751@pjwstk.edu.pl)	Issue Return
LT-5 (Laptop)	Piotr Natkaniec (s28145@pjwstk.edu.pl)	2026-04-26	2026-04-30	-	Overdue	Piotr Natkaniec (s28145@pjwstk.edu.pl)	Issue Return
URZ-4 (Monitor 24")	Piotr Natkaniec (s28145@pjwstk.edu.pl)	2026-06-21	2026-06-27	-	Active	Piotr Natkaniec (s28145@pjwstk.edu.pl)	Issue Return
SRV-3 (Macierz dyskowa)	Piotr Natkaniec (s28145@pjwstk.edu.pl)	2026-06-21	2026-06-27	-	Active	Piotr Natkaniec (s28145@pjwstk.edu.pl)	Issue Return
SW-2 (Licencja OS)	Piotr Natkaniec (s28145@pjwstk.edu.pl)	2026-06-21	2026-06-27	-	Active	Piotr Natkaniec (s28145@pjwstk.edu.pl)	Issue Return
URZ-1 (Monitor 24")	Tomasz Gajewski (s27751@pjwstk.edu.pl)	2026-06-22	2026-06-25	-	Active	Tomasz Gajewski (s27751@pjwstk.edu.pl)	Issue Return
AV-1 (Ekran projekcyjny)	Tomasz Gajewski (s27751@pjwstk.edu.pl)	2026-06-22	2026-06-25	-	Active	Tomasz Gajewski (s27751@pjwstk.edu.pl)	Issue Return

Rysunek 63 Zakładka Finalized Requests z widoczną tabelą wypożyczeń, kolorowymi oznaczeniami statusów oraz przyciskami Issue i Return w ostatniej kolumnie. Źródło: Opracowanie własne.

7.5. Incydynty

Obszar incydentów umożliwia rejestrowanie i obsługę problemów ze środkami trwałymi – uszkodzeń, awarii lub kradzieży. Dostęp do widoku operatorskiego posiada rola `incident_handler` oraz `ams_admin`, natomiast każdy użytkownik z rolą `ams_user` może zgłaszać własne incydynty i śledzić ich status przez sekcję widoków osobistych opisaną w podrozdziale 7.7.

7.5.1 Lista incydentów

Widok listy incydentów prezentowany jest w trybie tabelarycznym z wyszukiwaniem globalnym. Każdy wiersz tabeli zawiera tytuł incydentu, powiązany środek trwały, zgłaszającego użytkownika, priorytet, aktualny status oraz datę zgłoszenia. Priorytety incydentów oznaczone są trzema poziomami: Low, Medium i High. Formularz tworzenia incydentu zawiera pola: tytuł, opis, priorytet oraz wybór powiązanego środka trwałego. Rysunek 64 ilustruje widok listy incydentów.

Title	Asset	Reporter	Assigned To	Status	Priority	Created At	Closed At	Actions
Problem z portem HDMI	LT-27	Tomasz Gajewski (s27751@pjwstk.edu.pl)	-	Resolved	Medium	2026-04-23 05:07:09	2026-04-23 05:05:53	[icon] [icon]
Issue with PC	PC-1	Kacper Barański (s28166@pejot.edu.pl)	-	New	Medium	2026-06-10 16:15:28	-	[icon] [icon]

Rysunek 64 Widok listy incydentów w trybie tabelarycznym z widoczną kolumną priorytetu, statusu oraz przypisanego środka trwałego. Źródło: Opracowanie własne.

7.5.2 Widok szczegółów incydentu

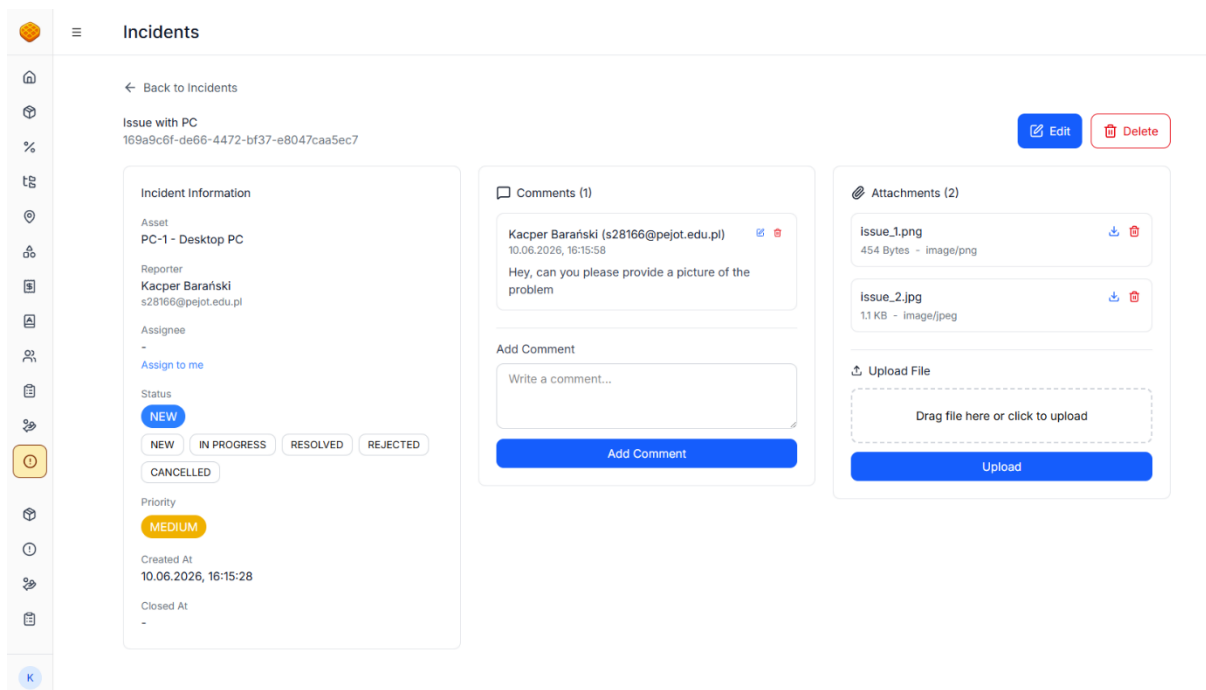
Widok szczegółów incydentu podzielony jest na trzy sekcje. Pierwsza sekcja, *Incident Information*, prezentuje pełne dane zgłoszenia - tytuł, opis, powiązany środek trwały, priorytet, datę zgłoszenia oraz aktualny status. Operator może zmieniać status incydentu bezpośrednio z poziomu tej sekcji – dostępne przyciski odpowiadają możliwym przejściom stanu, które przedstawia Tabela 19.

Tabela 19 Statusy incydentów i ich znaczenie. Źródło: Opracowanie własne.

Status	Znaczenie
NEW	Nowo utworzone zgłoszenie, oczekujące na akcje
IN PROGRESS	Incydent w trakcie rozwiązywania
RESOLVED	Incydent rozwiązany pomyślnie
REJECTED	Zgłoszenie odrzucone przez operatora
CANCELLED	Incydent anulowany przez zgłaszającego

Druga sekcja, *Comments*, implementuje wątek komentarzy powiązany z incydem. Każdy uprawniony użytkownik może dodać komentarz przez formularz tekstowy umieszczony na dole wątku. Autor komentarza może go edytować lub usunąć – dostępność tych akcji jest sprawdzana po stronie frontendu na podstawie identyfikatora autora z sesji. Komentarze prezentowane są chronologicznie z widoczną datą i autorem każdego wpisu.

Trzecia sekcja, *Attachments*, umożliwia dołączanie plików dokumentujących zdarzenie –zdjęć uszkodzeń, raportów serwisowych lub innych materiałów dowodowych. Mechanizm przesyłania plików oparty jest na podpisanych kryptograficznie adresach URL (ang. *pre-signed URLs*) generowanych przez SeaweedFS, opisanych szczegółowo w podrozdziale 5.6. W interfejsie użytkownik wybiera plik przez strefę przeciągania (ang. *drag and drop*) lub okno dialogowe przeglądarki. Po wybraniu pliku wyświetlana jest jego nazwa i rozmiar, a po potwierdzeniu następuje bezpośrednie przesłanie do magazynu obiektowego. Lista już dołączonych plików prezentuje nazwę pliku i jego typ MIME, a każdy załącznik można usunąć po potwierdzeniu w oknie dialogowym. Rysunek 65 prezentuje cały widok szczegółowy incydentu, wraz z trzema dostępnymi sekcjami.



Rysunek 65 Widok szczegółów incydentu z widocznymi sekcjami Incident Information, Comments oraz Attachments. Źródło: Opracowanie własne.

7.6. Personel

Widok personelu dostępny jest wyłącznie dla roli `ams_admin` i prezentuje listę wszystkich użytkowników systemu pobieranych z katalogu LDAP za pośrednictwem mikroserwisu `kc-proxy`. Widok jest tylko do odczytu – zarządzanie kontami odbywa się po stronie katalogu LDAP, a nie w aplikacji, zgodnie z założeniem opisanym w podrozdziale 5.4.

Lista personelu prezentowana jest wyłącznie w formie tabeli z wyszukiwaniem globalnym, a zaprezentowana została na Rysunku 66. Kolumny tabeli obejmują imię, nazwisko, adres e-mail oraz status konta (aktywne/nieaktywne). Widok nie udostępnia przycisków tworzenia, edycji ani usunięcia użytkownika – jedyną dostępną akcją jest przejście do widoku szczegółów danej osoby.

Personnel
Manage people responsible for assets

Search by...

Filters Columns

First Name	Last Name	E-mail	Enabled	Actions
Kacper	Barański	s28166@pejot.edu.pl	Yes	
Piotr	Natkaniec	s28145@pijwstk.edu.pl	Yes	
Tomasz	Gajewski	s27751@pijwstk.edu.pl	Yes	

Showing 4 records Rows per page 10 Page 1 of 1

Rysunek 66 Widok listy personelu w formie tabeli z kolumnami imienia, nazwiska, adresu e-mail i statusu konta. Źródło: Opracowanie własne.

Widok szczegółów użytkownika, zilustrowany na Rysunku 67, prezentuje jego pełny profil oraz zagregowane informacje o powiązanych z nim zasobach systemu w trzech sekcjach. Sekcja informacyjna zawiera imię, nazwisko, adres e-mail, status weryfikacji adresu, datę i godzinę utworzenia konta oraz przypisane grupy LDAP wyznaczające role systemowe użytkownika.

← Back to Personnel

Kacper Barański
kacper@example.pl

User Information

First Name

Kacper

Last Name

Barański

E-mail

s28166@pejot.edu.pl

E-mail Verified

No

Created

2025-06-10
17:05:06

Enabled

Yes

Groups

AMS ADMIN

AMS USER

ASSET MANAGER

INCIDENT HANDLER

INVENTORY AUDITOR

INVENTORY COUNTER

LOAN APPROVER

Rysunek 67 Widok szczegółów użytkownika z sekcją informacyjną. Źródło: Opracowanie własne.

Sekcja *Active Equipment Requests*, zaprezentowana na Rysunku 68, prezentuje tabelę aktywnych wypożyczeń przypisanych do danej osoby wraz z powiązanymi środkami trwałymi i datami wypożyczenia.

Active Equipment Requests (3)		
Search by... Columns		
Assets	Requested From	Requested To
LT-28 (Dell Inspiron 14u)	23.04.2026	30.04.2026
LT-27 (Asus ZenBook Pro)	23.04.2026	30.04.2026
URZ-1 (Monitor 24") URZ-1 (Monitor 24")	22.06.2026	25.06.2026

Showing 3 records

Rows per page: 5 Page 1 of 1

Rysunek 68 Sekcja Active Equipment Requests widoku szczegółów użytkownika wyświetlający wypożyczane środki trwałe, datę rozpoczęcia i datę zakończenia. Źródło: Opracowanie własne.

Sekcja *Assigned Assets* zawiera tabelę środków, za które użytkownik jest odpowiedzialny – kliknięcie wiersza przekierowuje bezpośrednio do widoku szczegółów danego środka. Rysunek 69 prezentuje sekcję środków trwałych przypisanych do użytkownika.

Assigned Assets (4)								
Search by... Filters Columns								
Asset Tag	Name	Type	Category	Assigned To	Condition	Net Price	Currency	
URZ-1	Monitor 24" Monitor 24 cale — Samsung	Monitor 24"	Urządzenia	s27751@pjwstk.edu.pl	Good	1 388,49	PLN (zł)	
AV-1	Ekran projekcyjny Ekran związany/stacjonarny — Samsung	Ekran projekcyjny	A/V	s27751@pjwstk.edu.pl	Needs Repair	952,41	PLN (zł)	
LT-27	Asus ZenBook Pro ASUS ZenBook Duo UX482EGR I7-1195G7/32...	Laptop	Sprzęt IT	s27751@pjwstk.edu.pl	Good	5 609,00	PLN (zł)	
LT-28	Dell Inspiron 14u Laptop Dell Inspiron 14 5440-7432 14" Core 7 ...	Laptop	Sprzęt IT	s27751@pjwstk.edu.pl	Good	2 708,00	EUR (€)	

Showing 4 records

Rows per page: 5 Page 1 of 1

Rysunek 69 Sekcja Assigned Assets widoku szczegółów użytkownika wyświetlający przypisane środki trwałe wyświetlający podstawowe informacje. Źródło: Opracowanie własne.

Sekcja *Assigned Campaigns*, przedstawiona na Rysunku 70, prezentuje kampanie inwentaryzacyjne, do których użytkownik został przypisany jako uczestnik, z możliwością przejścia do szczegółów każdej z nich.

Assigned campaigns (2)				
Search by... Filters Columns				
Code	Name	Start Date	End Date	Status
CAMP/2026/04/1	Kampania 1	24.04.2026	23.04.2026	Closed
CAMP/05/1	Kampania 1	29.05.2026	-	Active

Showing 2 records

Rows per page: 5 Page 1 of 1

Rysunek 70 Sekcja Assigned Campaigns widoku szczegółów użytkownika wyświetlający kampanie inwentaryzacyjne w których osoba bierze udział. Źródło: Opracowanie własne.

7.7. Widoki osobiste

Sekcja widoków osobistych grupuje widoki ograniczone wyłącznie do danych zalogowanego użytkownika. Są one dostępne dla wszystkich ról systemowych i stanowią podstawowy interfejs dla użytkowników z rolą `ams_user` oraz `inventory_counter`, którzy po zalogowaniu są automatycznie

przekierowywani do tej sekcji. Dane pobierane są przez dedykowane końcówki backendowe z przyrostkiem /my (np. /loan-requests/my, /incidents/my, /loans/my), które po stronie mikroserwisów filtrują wyniki do rekordów należących do zalogowanego użytkownika na podstawie identyfikatora z tokenu JWT.

7.7.1 Moje środki trwałe

Widok *My Assets* prezentuje środki trwałe, za które zalogowany użytkownik jest odpowiedzialny. Lista wyświetlana jest w formie tabeli z wyszukiwaniem globalnym oraz filtrami kolumnowymi dla kategorii i stanu technicznego. Kliknięcie wiersza przekierowuje do widoku szczegółów danego środka. Widok nie udostępnia żadnych akcji modyfikacji – służy wyłącznie do przeglądania przypisanych środków i ich stanu. Rysunek 71 ilustruje widok *My Assets*.

The screenshot shows the 'My Assets' page. At the top, there are three summary cards: 'Total Assets' with a value of 4, 'Active (Good)' with a value of 3, and 'In Maintenance (Needs Repair)' with a value of 1. Below these is a search bar and a table of assets. The table has columns: Asset Tag, Name, Type, Category, and Condition. There are four rows of assets. The first row is a monitor, the second is a keyboard, the third is a desktop PC, and the fourth is a mouse. Each row has a status button in the Category column: 'Urządzenia' for the monitor and keyboard, 'Sprzęt IT' for the PC, and 'Urządzenia' for the mouse. The Condition column shows 'Good' for the first three and 'Needs Repair' for the last one. At the bottom, it says 'Showing 4 records' and 'Rows per page 5'.

Asset Tag	Name	Type	Category	Condition
URZ-4	Monitor 24" Monitor 24 cale — Dell	Monitor 24"	Urządzenia	Good
URZ-12	Klawiatura Klawiatura przewodowa/bezprzewodowa — J...	Klawiatura	Urządzenia	Good
PC-1	Desktop PC Komputer stacjonarny — Dell	Desktop PC	Sprzęt IT	Good
URZ-2	Mysz Mysz komputerowa — HP	Mysz	Urządzenia	Needs Repair

Rysunek 71 Widok My Assets z tabelą środków trwałych przypisanych do zalogowanego użytkownika z widocznymi filtrami kolumnowymi kategorii i stanu technicznego. Źródło: Opracowanie własne.

7.7.2 Moje incydenty

Widok *My Incidents* prezentuje incydenty zgłoszone przez zalogowanego użytkownika, co zostało zaprezentowane na Rysunku 72. W odróżnieniu od operatorskiego widoku incydentów, użytkownik nie widzi zgłoszeń innych osób. Dostępny jest przycisk *Report Incident* otwierający formularz zgłoszenia nowego incydentu, identyczny z formularzem w widoku operatorskim. Użytkownik może edytować własne zgłoszenia oraz śledzić zmiany ich statusu wprowadzane przez operatora. Kliknięcie wiersza otwiera widok szczegółów incydentu z wątkiem komentarzy i listą załączników, opisany w podrozdziale 7.5.

My Incidents

Track and manage your reported incidents

[+ Report Incident](#)

Total Incidents: **4** New: **4** In Progress: **0** Rejected: **0** Resolved: **0**

Search by...

Filters Columns

Title	Asset	Assigned To	Status	Priority	Created At	Closed At	Actions
Issue with Monitor	URZ-4	-	New	High	2026-06-10 17:32:47	-	
Issue with Mouse	URZ-2	-	New	Medium	2026-06-10 17:32:31	-	
Issue with Keyboard	URZ-12	-	New	Low	2026-06-10 17:32:21	-	
Issue with PC	PC-1	-	New	Medium	2026-06-10 17:32:12	-	

Showing 4 records Rows per page: 5 Page 1 of 1

Rysunek 72 Widok **My Incidents** z tabelą własnych zgłoszeń, oznaczeniami priorytetu i statusu oraz przyciskiem **Report Incident**. Źródło: Opracowanie własne.

7.7.3 Moje wnioski o wypożyczenie

Widok *My Equipment Requests* jest uproszczoną wersją operatorskiego widoku wypożyczeń, podzieloną na dwie zakładki. Zakładka *Equipment Requests* prezentuje wnioski złożone przez zalogowanego użytkownika wraz z ich aktualnym statusem. Użytkownik może tworzyć nowe wnioski, edytować oczekujące oraz śledzić postęp rozpatrzenia. Zakładka *Finalized Requests* prezentuje aktywne wypożyczenia użytkownika – środki, które zostały mu wydane, z datami wypożyczenia i planowanego zwrotu oraz aktualnym statusem (SCHEDULED, ACTIVE, OVERDUE, RETURNED). Rysunek 73 przedstawia widok *My Equipment Requests* wraz z dostępnymi zakładkami.

My Equipment Requests

Track and manage your equipment requests

+ Request Equipment

Equipment Requests 3 Finalized Requests 4

Total Requests: 3, Received: 1, Rejected: 1, Pending: 1, Transferred Out: 0

Assets	Requested From	Requested To	Justification	Status	Comment
3	10.06.2026	13.06.2026	Example	Rejected	-
2	10.06.2026	13.06.2026	For video conference	Pending	-
4	10.06.2026	13.06.2026	Employee onboarding	Approved	-

Showing 3 records | Rows per page: 5 | Page 1 of 1

Rysunek 73 Widok My Equipment Requests z zakładkami Equipment Requests i Finalized Requests, tabelą wniosków z oznaczeniami statusów oraz przyciskiem tworzenia nowego wniosku. Źródło: Opracowanie własne.

7.7.4 Moje kampanie inwentaryzacyjne

Widok *My Inventory Campaigns* prezentuje kampanie, do których zalogowany użytkownik został przypisany jako uczestnik. Rysunek 74 przedstawia ten widok. Jest to podstawowy punkt wejścia dla roli *inventory_counter* – po zalogowaniu zostaje ona automatycznie przekierowana właśnie do tego widoku. Lista kampanii prezentuje nazwę, status oraz datę rozpoczęcia każdej z nich. Kliknięcie kampanii o statusie *ACTIVE* otwiera widok szczegółów z możliwością skanowania środków, analogiczny do widoku audytora opisanego w podrozdziale 7.3, jednak z ograniczonym zestawem dostępnych akcji – inwentaryzator nie może uruchamiać ani kończyć kampanii, a jedynie uczestniczyć w skanowaniu w jej trakcie.

8. Testowanie aplikacji

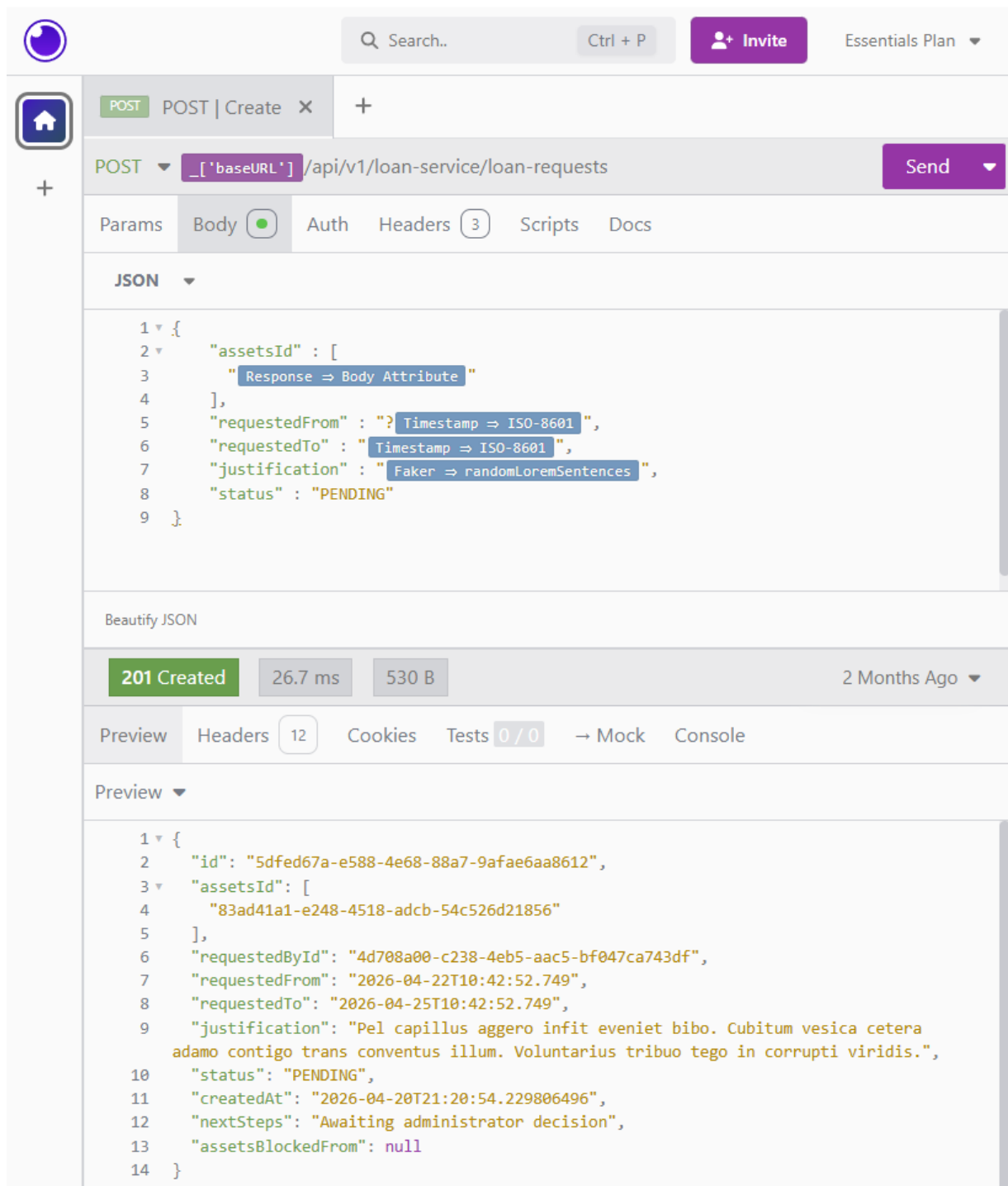
Niniejszy podrozdział opisuje proces weryfikacji poprawności działania aplikacji Glowing Waffle AMS. Z uwagi na prototypowy charakter pracy, a także ograniczenia czasowe wynikające z realizacji projektu inżynierskiego w niewielkim, 3 osobowym zespole, przyjęto pragmatyczną strategię testowania. Zdecydowane się na prowadzenie dwóch rodzajów testów; testów manualnych, prowadzonych przyrostowo wraz z pojawieniem się bądź modyfikacją istniejącej funkcjonalności. Dotyczy to zarówno końcówek po stronie backendu, jak i widoków użytkownika we frontendzie. Drugim typem testów były zautomatyzowane testy jednostkowe oraz testy pokrywające najbardziej złożoną logikę dziedziny systemu.

Należy uczciwie podkreślić, że testy napisane w ramach pracy nie mają absolutnie charakteru wyczerpującego. Świadomie zrezygnowano z dążenia do wysokiego pokrycia testami, na rzecz uzyskania większej ilości czasu. W szczególności szersze pokrycie testami integracyjnymi oraz E2E, obejmujące asynchroniczne przepływy między mikroserwisami, zostały ograniczone; trudność wiarygodnego, zautomatyzowanego testowania systemu rozproszonego wskazano jako jeden z problemów w podrozdziale 9.2. Tym samym, rozbudowę testów uwypuklono jako istotny kierunek dalszego rozwoju projektu.

8.1. Testy manualne

Podstawową formą bieżącej weryfikacji warstwy backendowej było manualne testowanie końcówek REST przy użyciu klienta API. Przyjęto zasadę, zgodnie z którą po każdym wdrożeniu nowej lub zmodyfikowanej końcówki jej działanie było sprawdzane ręcznie, jeszcze przed integracją z warstwą frontendową. W początkowej fazie prac wykorzystywano w tym celu narzędzie Postman, które następnie - z przyczyn opisanych w podrozdziale 4.14 - zastąpiono funkcjonalnie zbliżonym klientem Insomnia. Zespół utrzymywał współdzieloną kolekcję żądań, odwzorowującą zaimplementowane końcówki poszczególnych mikroserwisów, co pozwalało na powtarzalne wywoływanie operacji bez konieczności każdorazowego konstruowania żądania od podstaw u każdego z członków zespołu.

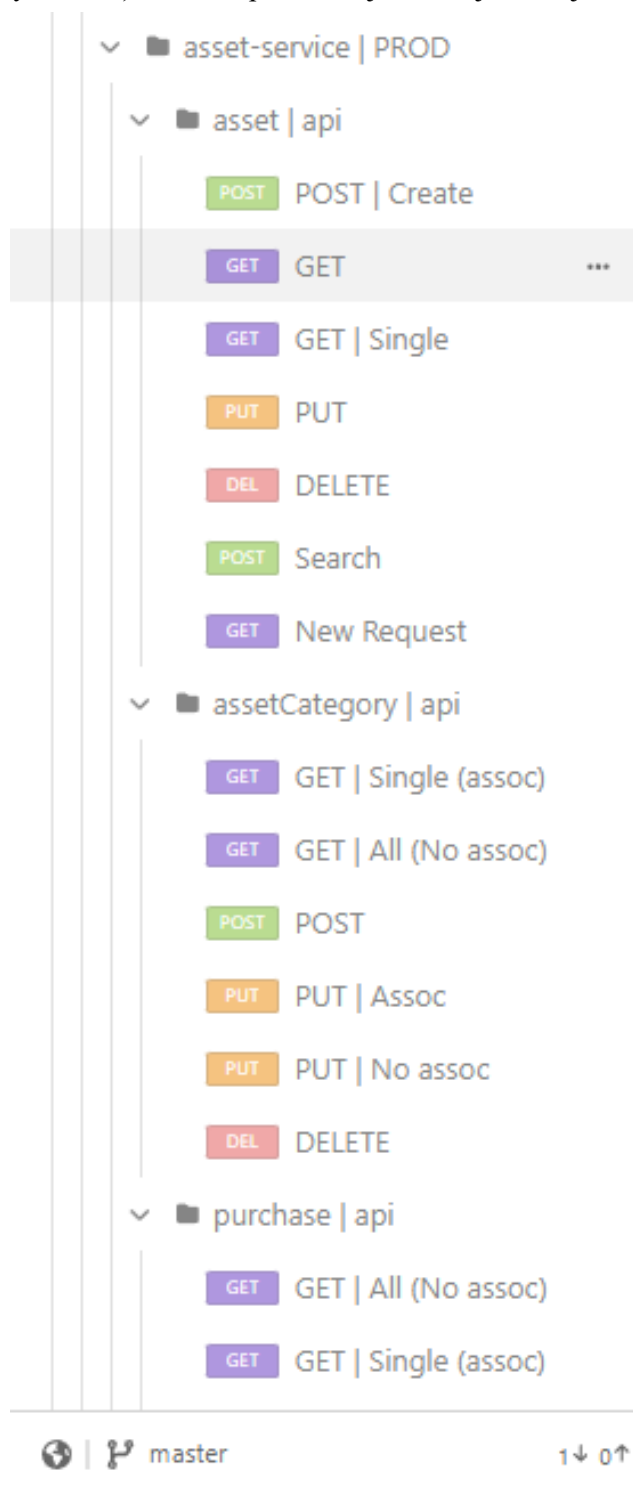
Manualna weryfikacja obejmowała sprawdzenie poprawności kodów odpowiedzi HTTP, struktury oraz zawartości zwracanych JSONów, a także reakcji końcówek na dane niepoprawne - przykładowo na naruszenie reguł walidacji danych wejściowych, czy próbę wykonania operacji bez wymaganych uprawnień. Warto zaznaczyć, że w nowoczesnych klientach API, takich jak Postman czy Insomnia, dużym ułatwieniem jest natywne wsparcie narzędzia dla standardu OAuth 2.0: konfiguracja przepływu *Authorization Code Flow* z rozszerzeniem PKCE na poziomie kolekcji (czyli zbioru żądań) pozwalała na automatyczne pozyskiwanie i dołączanie tokenu dostępu do nagłówka *Authorization*, bez ręcznego logowania przed każdym żądaniem. Do testowania scenariuszy, gdzie ciało żądania wymagało konkretnych danych, wykorzystano wbudowany mechanizm klienta Insomnia, który umożliwia generowanie testowych danych oraz pobierania autentycznych danych poprzez wykonanie innego żądania w locie. Zilustrowano to na Rysunku 75, gdzie pokazano generowanie losowego czasu w przyszłości i przeszłości (Timestamp => ISO-8601, z włączonymi opcjami), oraz faktycznego ciała odpowiedzi żądania z końcówki zwracającej ID środków trwałych (Response => Body Attribute), a także losowego tekstu typu Lorem Ipsum (Faker => randomLoremSentences).



Rysunek 75 Widok ciała żądania i odpowiedzi do końcówki loan-requests w Loan Svc. Źródło: Opracowanie własne.

Testowanie manualne uzupełniała interaktywna dokumentacja Swagger UI (omówiono w podrozdziale 4.16) umożliwiająca wywoływanie końcówek bezpośrednio z poziomu przeglądarki. Kolekcję żądań dla mikroserwisu Asset Service przedstawiono na Rysunku 76.

Warstwa frontendowa nie została objęta zautomatyzowanymi testami. Jej poprawność weryfikowano w sposób manualny, poprzez systematyczne przechodzenie przez interfejs użytkownika w przeglądarce (tzw. wyklikanie). Po implementacji każdej nowej funkcjonalności ręcznie



Rysunek 76 Część żądań w kolekcji Asset Service.

Źródło: Opracowanie własne.

przechodzono przez odpowiadające jej przepływy, sprawdzając zgodność zachowania aplikacji z oczekiwaniami. Weryfikacja obejmowała zarówno ścieżki podstawowe (ang. *happy path*), jak i scenariusze błędne - między innymi walidację formularzy, obsługę pustych stanów tabel, poprawność

komunikatów o błędach oraz mechanizm przedłużania sesji zrealizowany jako wyskakujący dymek. Sprawdzano także warunkowe renderowanie elementów interfejsu oraz ochronę tras w zależności od roli zalogowanego użytkownika, wykonując te same przepływy z kont o różnych zestawach uprawnień, zgodnie z modelem ról opisanym w rozdziale 7.

8.2. Testy zautomatyzowane

Jednostkowe, zautomatyzowane testy ograniczono do warstwy backendowej i zrealizowano przy użyciu biblioteki `spring-boot-starter-test`. Egzemplarz testu jednostkowego używającego atrapy serwisu `Loan Service`, tworzonej przez bibliotekę `Mockito`, zaprezentowano na Listingu 56.

Listing 56 Test jednostkowy weryfikujący poprawność rzucanego wyjątku. Źródło: Opracowanie własne.

```
@Test
void throwsWhenAssetIdsAreNull() {
    assertThatThrownBy(() -> service.findAvailability(null, FROM, TO))
        .assertInstanceOf(IllegalArgumentException.class)
        .hasMessageContaining("assetIds");
}
```

Jak wspomniano we wprowadzeniu do niniejszego rozdziału, zakres testów zautomatyzowanych pozostał świadomie ograniczony. Skoncentrowano je na najbardziej złożonej i obciążonej największym ryzykiem logice dziedzinowej - regułach dostępności środków w mikroserwisie wypożyczeń.

9. Podsumowanie i wnioski

Niniejszy podrozdział stanowi podsumowanie prac nad systemem Glowing Waffle Asset Management System i jednocześnie zamyka przyjętą w rozdziale 5 tezę świadomego rachunku kompromisów architektonicznych. Rezultatem jest w pełni funkcjonalny prototyp, które założenia, architekturę oraz implementację opisano w rozdziałach 3-6, a interfejs użytkownika i proces testowania odpowiedzi w rozdziałach 7 oraz 8. Należy przy tym podkreślić, że zgodnie z deklaracją przyjętą w podrozdziale 1.1, celem pracy nie było dostarczenie gotowego produktu komercyjnego, lecz wykazanie zasadności oraz ustalenie wykonalności przyjętych założeń oraz koncepcji architektury systemu.

9.1. Ocena spełnienia wymagań funkcjonalnych i нефункциональных

9.1.1 Spełnialność wymagań funkcjonalnych

Wymagania funkcjonalne systemu Glowing Waffle AMS przedstawiono w podrozdziale 3.5 w postaci dziewięciu tabel (Tabele 3-11). Obejmują siedem grup funkcji inicjowanych przez użytkownika oraz dwie grupy działań inicjowanych automatycznie przez system. Łącznie daje to 73 wymagania; ich spełnienie zweryfikowano poprzez odwzorowanie każdego wymagania na odpowiadającą mu funkcję prototypu, co zademonstrowano w interfejsie użytkownika w rozdziale 7 i objęto testami manualnymi opisanymi w rozdziale 8. Celem nie powielania zamieszczonych tam rysunków, ocenę w tym podrozdziale przedstawiono w formie zbiorczej, w podziale na obszary specyfikacji oraz realizujące je mikroserwisy, w Tabeli 20.

Tabela 20 Zbiorcza ocena spełnienia wymagań funkcjonalnych w podziale na obszary specyfikacji z podrozdziału 3.5. Źródło: Opracowanie własne.

Wymagania funkcjonalne	Tabela	Liczba	Mikroserwis/y realizujący/e	Spełnialność
Zarządzanie środkami trwałymi (ŚT)	Tab. 3	7	Asset Svc, Reporting Svc, Notification Svc	TAK
Klasyfikacja ŚT i pola niestandardowe	Tab. 4	11	Asset Svc	TAK
Zarządzanie lokalizacjami	Tab. 5	4	Asset Svc	TAK
Finansowa ewidencja zasobów	Tab. 6	8	Asset Svc	TAK
Wypożyczenia	Tab. 7	9	Loan Svc	TAK
Wypożyczenia – działania automatyczne	Tab. 8	5	Loan Svc, Notification Svc, Inventory Svc	TAK

Inwentaryzacja ŚT	Tab. 9	11	Inventory Svc	TAK
Inwentaryzacja – działania automatyczne	Tab. 10	5	Inventory Svc, Reporting Svc	TAK
Obsługa incydentów	Tab. 11	13	Incident Svc	TAK

Wszystkie wymagania funkcjonalne zestawione w podrozdziale 3.5 zostały zrealizowane w prototypie; zaimplementowano pełen cykl życia ŚT – ewidencję opartą na centralnym, elastycznym modelu danych z hierarchią kategoria-typ, dziedziczeniem pól w kategoriach i typach, finansową ewidencję faktur i stawek VAT, zarządzanie lokalizacjami, obsługę wypożyczeń i zwrotów oraz obsługę incydentów wraz z komentarzami i załącznikami. Wymaganie elastyczności modelu danych, bezpośrednio zawarte w tytule tej pracy inżynierskiej, odwzorowano w całości. Stanowi to główny punkt wyróżniający prototyp na tle konkurencyjnych rozwiązań.

9.1.2 Spełnialność wymagań niefunkcjonalnych

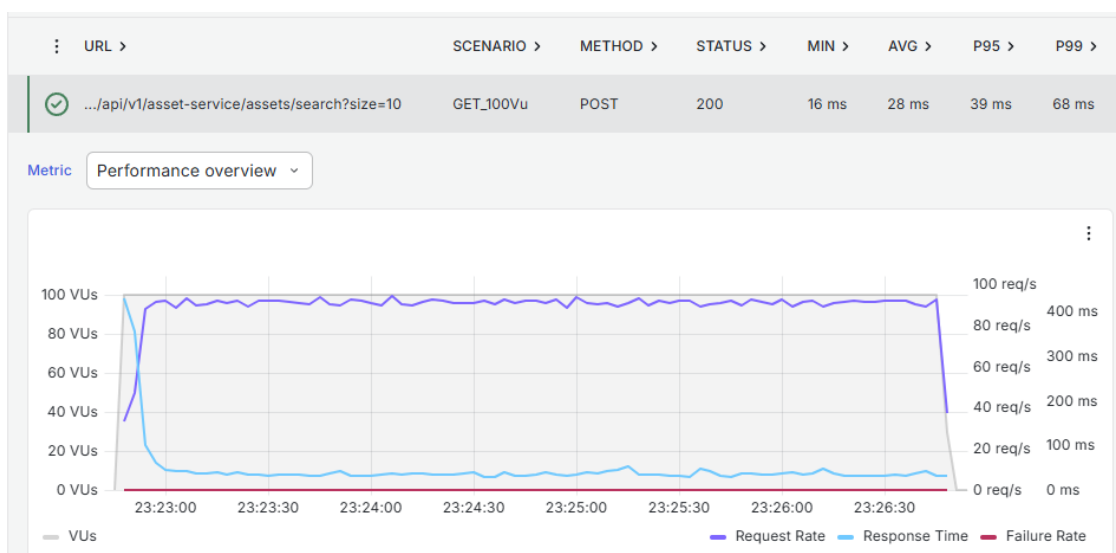
Do zweryfikowania części wymagań niefunkcjonalnych posłużono się maszyną wirtualną stanowiącą docelowe środowisko produkcyjne dla prototypu; instancję w chmurze Hetzner o parametrach 8 wirtualnych rdzeni, 16 GB pamięci ram oraz 160GB pamięci dyskowej. Na maszynie tej współdzielone są wszystkie komponenty systemu, tj. uruchomione są wszystkie elementy pliku produkcyjnego Docker Compose. Pomiar odniesiono do SLO zdefiniowanych bezpośrednio w podrozdziale 3.8. Baza danych mikroservisu Asset Service została wypełniona reprezentatywną liczbą 10 tysięcy środków trwałych.

Jako narzędzie do obciążenia wykorzystano k6, uruchomione na infrastrukturze Grafana Cloud k6, ze strefy obciążeniowej AWS Frankfurt. Komunikacja z bramą API odbywała się po protokole HTTP/2 z szyfrowaniem TLS 1.3. Scenariusz testowy utrzymywał stałe obciążenie 100 współbieżnych użytkowników wirtualnych (ang. *virtual users*, *VU*) przez 4 minuty. Pojedyncza iteracja użytkownika obejmowała sześć żądań do reprezentatywnych końcówek pięciu mikroservisów. Wskaźnikiem SLI był czas odpowiedzi żądania, mierzony po stronie klienta (czyli generatora obciążenia) i agregowany do percentyli P90, P95 oraz P99. Całkowita liczba żądań do każdej końcówki wynosiła 21,700, a maksymalne nasycenie żadaniami na sekundę (ang. *requests per second*, *req/s*) wyniosło 564,67. Szczegółowe wyniki testu przedstawiono w Tabeli 21. Wszystkie czasy podano w milisekundach.

Tabela 21 Rozkład czasu odpowiedzi końcówek reprezentatywnych przy obciążeniu 100 VU (strefa AWS Frankfurt, czas trwania 4 min, 21700 żądań na końcówkę). Wartości w milisekundach. Źródło: Opracowanie własne.

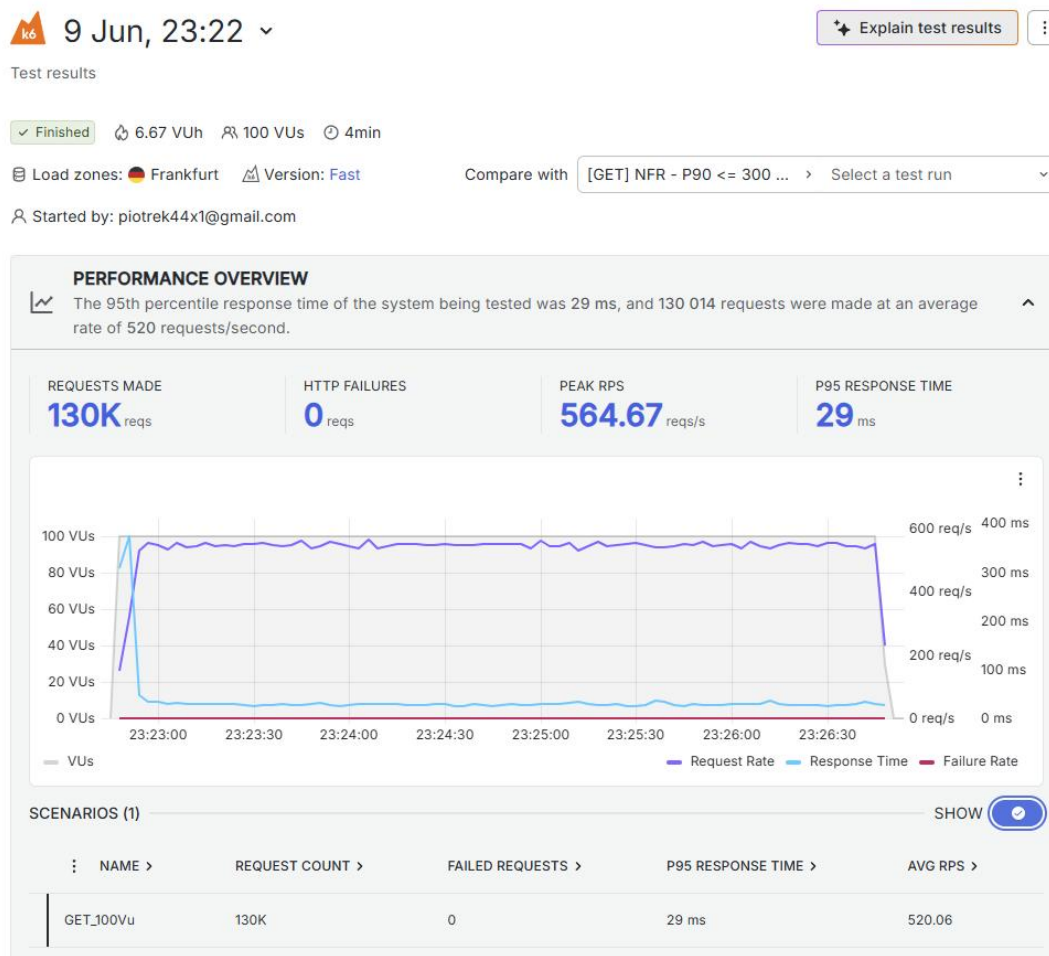
Końcówka (mikroserwis)	Min	Avg	P95	P99	Spełnialność SLO
GET loan-requests (Loan Svc)	13	23	31	40	TAK
GET campaigns (Inventory Svc)	11	17	22	29	TAK
GET incidents (Incident Svc)	7.9	11	14	19	TAK
POST assets-search (Asset Svc)	16	28	39	68	TAK
GET asset-categories (Asset Svc)	8.2	12	16	28	TAK

Wykres *VU*, czasów odpowiedzi, współczynnika błędów i żądań dla końcówki wyszukującej środki trwale w mikroserwisie Asset Service przedstawiono na Rysunek 77.



Rysunek 77 Wykres przedstawiający wirtualnych użytkowników, współczynnik błędów, czasów odpowiedzi na osi czasu dla końcówki POST asset. Źródło: Opracowanie własne.

Panel podsumowujący cały test przedstawiono na Rysunek 78.



Rysunek 78 Podsumowanie testu wydajnościowego infrastruktury backendowej systemu Glowing Waffle AMS. Źródło: Opracowanie własne.

Uzyskane wyniki wskazują na spełnienie wymagań niefunkcjonalnych dotyczących opóźnień ze znacznym zapasem. Najwyższy zaobserwowany dla końcówek GET percentyl P99 wyniósł 40 ms i jest ponad siedemnastokrotnie niższy od przyjętego celu 700 ms, natomiast nawet P95 (do 31 ms) pozostaje blisko dziesięciokrotnie poniżej celu wyznaczonego dla P90 (≤ 300 ms).

9.2. Napotkane problemy

W trakcie prac nad prototypem zespół napotkał pewne trudności o charakterze zarówno koncepcyjnym, jak i technicznym. Uznano, że ich uczciwe odnotowanie będzie miało wartość dodaną, gdyż są to problemy nierozzerwalnie związane z konsekwencjami przyjętych decyzji. Poniżej wypisano najważniejsze trudności:

- Złożoność operacyjna i obliczeniowa - mikroserwisowy charakter systemu sprawił, że uruchomienie środowiska deweloperskiego, lokalnie na laptopie programisty wymaga zestawienia aż 26 kontenerów – sześciu mikroserwisów domenowych, każdy z odrębną instancją bazy danych PostgreSQL, brokera RabbitMQ, serwera Keycloak, magazynu S3 SeaweedFS a także komponentów związanych ze stosem obserwowalności, takich jak OpenTelemetry Collector, Grafana czy Jaeger bądź Prometheus. Przełożyło się to na wysokie zużycie zasobów

oraz wydłużony czas startu; w konsekwencji, okazało się niemożliwym rozwijać system na bieżąco bez odpowiednio mocnego środowiska lokalnego. Kompletną listę kontenerów obecnych na środowisku deweloperskim zilustrowano na Rysunek 79.

☐ Only show running containers

☐	Name	Image	Port(s)	Actions
☐	otel-collector	otel/opentelemetry-collector-contrib:0.148	4317:4317 Show all ports (2)	
☐	inventory-service-1	glowing-waffle-inventory-service	8082:8080 Show all ports (2)	
☐	asset-service-1	glowing-waffle-asset-service	8081:8080 Show all ports (2)	
☐	loan-service-1	glowing-waffle-loan-service	8083:8080 Show all ports (2)	
☐	incident-service-1	glowing-waffle-incident-service	8085:8080 Show all ports (2)	
☐	reporting-service-1	glowing-waffle-reporting-service	8087:8080 Show all ports (2)	
☐	loki	grafana/loki:3.4.2	3100:3100 Show all ports (2)	
☐	notification-service-1	glowing-waffle-notification-service	8084:8080 Show all ports (2)	
☐	inventory-service-db	postgres:15.17-alpine	5401:5432 Show all ports (2)	
☐	grafana-1	grafana/grafana:11.5.2		
☐	asset-service-db	postgres:15.17-alpine	5400:5432 Show all ports (2)	
☐	loan-service-db	postgres:15.17-alpine	5402:5432 Show all ports (2)	
☐	incident-service-db	postgres:15.17-alpine	5404:5432 Show all ports (2)	
☐	jaeger	jaegertracing/all-in-one:1.76.0		
☐	reporting-service-db	postgres:15.17-alpine	5405:5432 Show all ports (2)	
☐	notification-service-db	postgres:15.17-alpine	5403:5432 Show all ports (2)	
☐	rabbitmq-1	rabbitmq:4.2.5-management		
☐	kc-proxy-1	glowing-waffle-kc-proxy	8086:8080 Show all ports (2)	
☐	prometheus	prom/prometheus:v3.10.0	9090:9090 Show all ports (2)	
☐	ams-object-storage	chrislusf/seaweedfs:4.17		
☐	openldap	osixia/openldap:1.5.0		
☐	keycloak	keycloak/keycloak:26.5.6	3200:8080 Show all ports (2)	
☐	frontend-service-1	glowing-waffle-frontend-service		
☐	phpldapadmin	osixia/phpldapadmin:0.9.0		
☐	traefik	traefik:v3.6.11		

Showing 26 items

Rysunek 79 Zbiór wszystkich kontenerów Docker na środowisku deweloperskim. Źródło: Opracowanie własne.

- Ograniczenia darmowej wersji bramy Traefik w zakresie weryfikacji JWT – w toku prac okazało się, iż Traefik w wersji *community* nie wspiera sprawdzania tokenów na poziomie żądania. W konsekwencji, podjęto decyzję o przeniesieniu tej odpowiedzialności do każdego mikroservisu z

osobna, a także połączenie konfiguracji Spring Security z końcówkami JWKS Keycloak. Jest to odstępstwo od tego co planowano pierwotnie, czyli uwierzytelnienia na brzegu systemu.

- Konsekwencje wzorca *Database per Service* - brak możliwości wykonania zapytania łączącego dane z wielu kontekstów dziedzinowych za pomocą klauzuli JOIN wymusił realizację takich operacji na poziomie warstwy aplikacyjnej, czerpiąc częściowo ze wzorca *API Composition*, który opisano w podrozdziale 5.5. W praktyce oznaczało to więcej kodu oraz konieczność starannego projektowania zakresu danych redundantnych pomiędzy serwisami. Co więcej, jak opisano (również w podrozdziale 5.5), w aplikacjach dla dużych przedsiębiorstw powinno się wprowadzić celową, zwielokrotnioną redundancję danych między mikroservisami, gdzie serwis X przechowuje pewne dane serwisu Y celem uniknięcia narzutu sieciowego, który wynikałby z ciągłego odpytywania Y przez X. W prototypie natomiast zdecydowano się nie wprowadzać aż tak szeroko zakrojonej redundancji, która jedynie zwiększałaby i tak już znaczną złożoność systemu.
- Testowanie systemu rozproszonego. Wiarygodne, zautomatyzowane testowanie przepływów asynchronicznych obejmujących wiele mikroservisów jest trudniejsze niż testowanie aplikacji monolitycznej. W rezultacie zautomatyzowane testy jednostkowe skoncentrowano na najbardziej złożonej logice dziedzinowej, w szczególności na regułach dostępności i decyzji w mikroservisie wypożyczeń, natomiast szersze pokrycie testami integracyjnymi oraz end-to-end pozostało ograniczone. Zagadnienie to rozwinęto w rozdziale 8 i wskazano jako istotny kierunek dalszego rozwoju w podrozdziale 9.3.
- Zmiana decyzji dotyczącej magazynu plików typu S3 self-hosted - pierwotnie rozważanym magazynem obiektowym było rozwiązanie MinIO. Niestety, zmiany w jego modelu licencyjnym [52] oraz zubożenie panelu administracyjnego w wersji społecznościowej [53] wymusiły zmianę decyzji i ostateczny wybór rozwiązania SeaweedFS (podrozdział 5.6).
- Integracja z serwerem tożsamości - poprawne zestawienie przepływu Authorization Code Flow z rozszerzeniem PKCE, mapowania ról z LDAP na grupy Keycloak a później na *Claims* tokenu JWT, dedykowanego konwertera ról Keycloak oraz konfiguracji *CORS* okazało się czasochłonne i wymagało sporej analizy. Dodatkowo, następstwem krótkiego czasu życia tokenu JWT była konieczność zaprojektowania sensownego mechanizmu przedłużania sesji. Zostało to zrealizowane jako wyskakujący dymek w interfejsie użytkownika, co opisano w podrozdziale 5.4

9.3. Dalszy rozwój

Prototypowy charakter systemu w naturalny sposób wyznacza obszar, w którym platforma może się rozwinąć. Proponowane kierunki pogrupowano tematycznie. Wiele z wymienionych pozycji stanowi bezpośrednią odpowiedź na ograniczenia oraz ryzyka zestawione w podrozdziale 9.2.

- Niezawodność i wysoka dostępność – priorytetowym kierunkiem rozwoju jest przeniesienie platformy produkcyjnej na dojrzały orkiestrator kontenerów klasy Kubernetes, uruchomienie mikroservisów w wielu replikach oraz wdrożenie komponentów infrastrukturalnych, takich jak Traefik, Keycloak, PostgreSQL czy RabbitMQ, w trybie wysokiej dostępności (*High Availability*). Wszystkie mikroservisy zaprojektowane zostały z myślą o byciu w 100% bezstanowymi, zdolnymi do replikacji horyzontalnej w dowolnej liczbie. Dodatkowo, do systemu powinien zostać wprowadzony wzorzec bezpiecznika (ang. *Circuit Breaker*), co poruszono szerzej w rozdziale 5.3.
- Spójność danych - należy wdrożyć wzorzec *Transactional Outbox* oraz konsekwentnie zapewnić idempotencję dla mikroservisów konsumujących zdarzenia asynchroniczne, na przykład poprzez ewidencję identyfikatorów już przetworzonych komunikatów. Kwestie te zasygnalizowane zostały w podrozdziałach 5.2 oraz 5.5.

- Zarządzanie schematem bazy danych - należy rozważyć wprowadzenie migracji bazodanowych, np. przy użyciu rozwiązania Flyway, celem zapewnienia wersjonowania zmian zachodzących w encjach bazodanowych [44] (str. 199).
- Rozszerzenia funkcjonalne. Naturalnymi kierunkami rozbudowy dziedziny są: moduł amortyzacji i integracji z zewnętrznymi systemami księgowymi (co zbliżyłoby system do klasy FAMS, opisanej w rozdziale 2), warunki świadczenia usług (ang. SLA) dla incydentów, rozbudowa widoków raportowych w oparciu o zdarzenia gromadzone przez Reporting Service. Niemniej, bazując na ocenie stanu sztuki i istniejących rozwiązań, nie sposób opisać pełnej listy możliwych kierunków rozwoju funkcjonalnego z uwagi na duże skomplikowanie dziedziny biznesowej.

Przedstawiona lista nie jest kompletna, wyznacza jedynie propozycję ścieżkę rozwoju prototypu w kierunku docelowego systemu produkcyjnego, w którym każdy z wymienionych kroków adresuje konkretne, zidentyfikowane wcześniej ograniczenie.

9.4. Wnioski końcowe

Przeprowadzone prace pozwoliły zrealizować cel główny pracy, którym było zaprojektowanie i zaimplementowanie elastycznego oraz skalowalnego systemu administracyjno-inwentaryzacyjnego. Powstały prototyp systemu Glowing Waffle AMS obejmuje pełen zaplanowany zakres; ewidencję środków trwałych z elastycznym modelem danych opartym na hierarchii kategoria–typ oraz polach niestandardowych, finansową ewidencję faktur i stawek VAT, zarządzanie lokalizacjami, obsługę wypożyczeń, kampanii inwentaryzacyjnych oraz incydentów. Tym samym potwierdzona jest wykonalność koncepcji nakreślonej w rozdziale 3.

Najważniejszym wnioskiem inżynierskim, wynikającym z całości prac, jest praktyczne potwierdzenie tezy otwierającej rozdział 5 - że każda decyzja architektoniczna jest kompromisem. Architektura mikroserwisowa dostarczyła wymiernych korzyści w postaci luźnego powiązania komponentów (zgodnie z *Domain Driven Design*), niezależnego wdrażania oraz potencjału skalowalności i odporności na awarie, jednak wprowadziła duże koszty: utratę atomowości operacji rozproszonych, konieczność zapewnienia spójności poprzez wzorzec sagi oraz znaczny wzrost złożoności operacyjnej i diagnostycznej. W odniesieniu do skali samego prototypu koszty te lokują się blisko granicy nadmiernej inżynierii (ang. *overengineering*); w rzeczywistym systemie produkcyjnym ich poniesienie powinno być jednoznacznie uzasadnione skalą organizacji, liczebnością zespołów oraz wymaganiami niefunkcjonalnymi. Jednocześnie, z perspektywy edukacyjnej i badawczej, świadome zderzenie się z tymi kompromisami i złożonością zapewniło każdemu członkowi zespołu z osobna ogromną wartość. Należy również uwypuklić, iż praca w zespole stanowi świetny trening umiejętności miękkich i przywódczych, w szczególności w zakresie wypracowywania jednolitego stanowiska w grupie.

10. Bibliografia

- [1] ; Atlassian, „What is IT asset management?,” [Online]. Available: <https://www.atlassian.com/itsm/it-asset-management>. [Data uzyskania dostępu: 05 06 2026].
- [2] ; „Snipe-IT Introduction,” [Online]. Available: <https://snipe-it.readme.io/docs/introduction>. [Data uzyskania dostępu: 05 06 2026].
- [3] ; Virima, „Top Open Source ITAM Software ranking,” [Online]. Available: <https://virima.com/blog/top-open-source-it-asset-management-software>. [Data uzyskania dostępu: 05 06 2026].
- [4] ; GLPI, „GLPI Documentation,” [Online]. Available: <https://help.glpi-project.org/documentation/readme-1-1/general>. [Data uzyskania dostępu: 06 06 2026].
- [5] ; EZO, „EZO Inventory Features,” [Online]. Available: <https://ezo.io/ezofficeinventory/features/>. [Data uzyskania dostępu: 07 06 2026].
- [6] ; AssetPanda, "Asset Pando PRO Documentation," [Online]. Available: <https://prohelp.assetpanda.com/>. [Accessed 07 06 2026].
- [7] ; Google, „SRE fundamentals: SLIs, SLAs and SLOs,” [Online]. Available: <https://cloud.google.com/blog/products/devops-sre/sre-fundamentals-slis-slas-and-slos>. [Data uzyskania dostępu: 09 06 2026].
- [8] ; C. S. Horstmann, Java. Podstawy. Wydanie XI, Gliwice: Helion SA, 2019. ISBN 9788328357785.
- [9] ; C. Walls, Spring w akcji. Wydanie V, Gliwice: Helion SA, 2019. ISBN 9788328903357.
- [10] ; M. Trzaska, Modelowanie i implementacja systemów informatycznych 2.0., 2025. ISBN 9788397644205.
- [11] ; Hibernate Team, „What is object/relational mapping?,” Hibernate, [Online]. Available: <https://hibernate.org/orm/what-is-an-orm/>. [Data uzyskania dostępu: 16 05 2026].
- [12] ; The Apache Software Foundation, „What is Maven?,” The Apache Software Foundation, [Online]. Available: <https://maven.apache.org/what-is-maven.html>. [Data uzyskania dostępu: 16 05 2026].
- [13] ; The PostgreSQL Global Development Group, „About PostgreSQL,” The PostgreSQL Global Development Group, [Online]. Available: <https://www.postgresql.org/about/>. [Data uzyskania dostępu: 16 05 2026].
- [14] ; Microsoft, „microsoft/TypeScript,” GitHub, [Online]. Available: <https://github.com/microsoft/TypeScript>. [Data uzyskania dostępu: 19 05 2026].
- [15] ; Microsoft, „TypeScript: JavaScript With Syntax For Types,” Microsoft, [Online]. Available: <https://www.typescriptlang.org/>. [Data uzyskania dostępu: 19 05 2026].
- [16] ; Microsoft, „The TypeScript Handbook,” Microsoft, [Online]. Available: <https://www.typescriptlang.org/docs/handbook/intro.html>. [Data uzyskania dostępu: 19 05 2026].
- [17] ; Microsoft, „Everyday Types,” Microsoft, [Online]. Available: <https://www.typescriptlang.org/docs/handbook/2/everyday-types.html>. [Data uzyskania dostępu: 19 05 2026].
- [18] ; Microsoft, „Type Inference,” Microsoft, [Online]. Available: <https://www.typescriptlang.org/docs/handbook/type-inference.html>. [Data uzyskania dostępu: 19 05 2026].
- [19] ; Vercel, „Next.js Docs,” Vercel, [Online]. Available: <https://nextjs.org/docs>. [Data uzyskania dostępu: 19 05 2026].

[20] ; Broadcom, „RabbitMQ,” Broadcom, [Online]. Available: <https://www.rabbitmq.com/>. [Data uzyskania dostępu: 07 06 2026].

[21] ; Broadcom, „RabbitMQ Reliability Guide,” VMWare, [Online]. Available: <https://www.rabbitmq.com/docs/reliability>. [Data uzyskania dostępu: 10 05 2026].

[22] ; Broadcom, „RabbitMQ with Spring AMQP,” [Online]. Available: <https://www.rabbitmq.com/tutorials/tutorial-six-spring-amqp#putting-it-all-together>. [Data uzyskania dostępu: 27 05 2026].

[23] ; D. Inc., „What is Docker?,” Docker Inc., [Online]. Available: <https://docs.docker.com/get-started/docker-overview/>. [Data uzyskania dostępu: 20 05 2026].

[24] ; Traefik Labs, „Traefik Proxy: Modern HTTP Reverse Proxy and Load Balancer,” Traefik Labs, [Online]. Available: <https://doc.traefik.io/traefik/>. [Data uzyskania dostępu: 20 05 2026].

[25] ; Keycloak, „Keycloak Documentation,” Cloud Native Computing Foundation, [Online]. Available: <https://www.keycloak.org/documentation>. [Data uzyskania dostępu: 24 05 2026].

[26] ; B. S. Scott Chacon, „About Git,” Git, [Online]. Available: <https://git-scm.com/about>. [Data uzyskania dostępu: 25 05 2026].

[27] ; B. S. Scott Chacon, „Pro Git Book - Getting Started - About Version Control,” Git, [Online]. Available: <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>. [Data uzyskania dostępu: 25 05 2026].

[28] ; GitHub, „Understanding GitHub Actions,” GitHub, [Online]. Available: <https://docs.github.com/en/actions/get-started/understand-github-actions>. [Data uzyskania dostępu: 25 05 2026].

[29] ; Postman, „Postman Docs,” Postman Inc., [Online]. Available: <https://learning.postman.com/>. [Data uzyskania dostępu: 19 05 2026].

[30] ; Postman, „Postman pricing,” Postman Inc., [Online]. Available: <https://www.postman.com/pricing/>. [Data uzyskania dostępu: 19 05 2026].

[31] ; Insomnia, „Insomnia Docs,” Kong Inc., [Online]. Available: <https://developer.konghq.com/insomnia/>. [Data uzyskania dostępu: 20 05 2026].

[32] ; Discord, „Discord,” Discord Inc., [Online]. Available: <https://discord.com/>. [Data uzyskania dostępu: 20 05 2026].

[33] ; Redwerk, „MyRepoBot - Documentation,” Redwerk, [Online]. Available: <https://myrebot.com/faq>. [Data uzyskania dostępu: 20 05 2026].

[34] ; S. Software, „What Is OpenAPI?,” SmartBear Software, [Online]. Available: https://swagger.io/docs/specification/v3_0/about/. [Data uzyskania dostępu: 25 05 2026].

[35] ; OpenTelemetry, „What is OpenTelemetry?,” OpenTelemetry, [Online]. Available: <https://opentelemetry.io/docs/what-is-opentelemetry/>. [Data uzyskania dostępu: 25 05 2025].

[36] ; SeaweedFS, „seaweedfs/seaweedfs,” GitHub, [Online]. Available: <https://github.com/seaweedfs/seaweedfs>. [Data uzyskania dostępu: 26 05 2026].

[37] ; Prometheus Authors, „Overview,” Prometheus, [Online]. Available: <https://prometheus.io/docs/introduction/overview/>. [Data uzyskania dostępu: 26 05 2026].

[38] ; G. Labs, „Introduction,” Grafana Labs, [Online]. Available: <https://grafana.com/docs/grafana/latest/fundamentals/>. [Data uzyskania dostępu: 26 05 2026].

[39] ; Jaeger Authors, „Introduction,” Cloud Native Computing Foundation, [Online]. Available: <https://www.jaegertracing.io/docs/2.18/>. [Data uzyskania dostępu: 26 05 2026].

[40] ; Grafana Labs, „Get started with Grafana Loki,” Grafana Labs, [Online]. Available: <https://grafana.com/docs/loki/latest/get-started/>. [Data uzyskania dostępu: 26 05 2026].

[41] ; F. N. Richards M, Fundamentals of Software Architecture: An Engineering Approach, Sebastopol: O'Reilly Media, 2020. ISBN 9781492043454.

- [42] ; E. Evans, Domain-Driven Design: Tackling Complexity in the Heart, Boston: Addison-Wesley Professional, 2003. ISBN 9780321125217.
- [43] ; C. Richardson, Microservices Patterns: With Examples in Java, Shelter Island: Manning Publications, 2019. ISBN 9781617294549.
- [44] ; C. Harris, „Microservices vs. monolithic architecture,” Atlassian, [Online]. Available: <https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith>. [Data uzyskania dostępu: 05 05 2026].
- [45] ; R. T. Fielding, „Architectural Styles and the Design of Network-based Software Architectures,” 2000. [Online]. Available: https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf. [Data uzyskania dostępu: 2026 05 15].
- [46] ; Amazon, „Authenticating Requests (AWS Signature Version 4),” [Online]. Available: <https://docs.aws.amazon.com/AmazonS3/latest/API/sigv4-query-string-auth.html>.
- [47] ; C. Sridharan, Distributed Systems Observability: A Guide to Building Robust Systems, 2018. ISBN 9781492033400.
- [48] ; Cloud Native Project Foundation, „OpenTelemetry Collector Architecture,” [Online]. Available: <https://opentelemetry.io/docs/collector/architecture/>. [Data uzyskania dostępu: 30 05 2026].
- [49] ; Google, „Distroless Images,” [Online]. Available: <https://github.com/googlecontainertools/distroless>. [Data uzyskania dostępu: 09 06 2026].
- [50] ; A. Polak, „Honey, I shrunk the node modules! ...and improved app performance,” [Online]. Available: <https://tsh.io/blog/reduce-node-modules-for-better-performance>. [Data uzyskania dostępu: 09 06 2026].
- [51] ; Renato Losio, „MinIO GitHub Repository in Maintenance Mode: What's Next for the Open Source Object Storage?,” [Online]. Available: <https://www.infoq.com/news/2025/12/minio-s3-api-alternatives/>. [Data uzyskania dostępu: 09 06 2026].
- [52] ; Cloudian, „MinIO Bait and Switch Leaves Organizations Reeling for Alternatives,” [Online]. Available: <https://cloudian.com/blog/minios-ui-removal-leaves-organizations-searching-for-alternatives/>. [Data uzyskania dostępu: 09 06 2026].
- [53] ; M. Dick Hardt, „Internet Engineering Task Force (IETF) - RFC6749 - The OAuth 2.0 Authorization Framework,” 10 2012. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc6749>. [Data uzyskania dostępu: 16 05 2026].
- [54] ; Atlassian, „Git Branching Strategies,” [Online]. Available: <https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow>. [Data uzyskania dostępu: 07 06 2026].

11. Spis ilustracji

Rysunek 1 Główny widok repozytorium prototypu Glowing Waffle AMS na platformie GitHub. Źródło: Opracowanie własne.	9
Rysunek 2 Lista niektórych gałęzi rozwijania kodu, które powstały podczas tworzenia prototypu. Źródło: Opracowanie własne.	10
Rysunek 3 Powiadomienie informujące o nowym, otwartym PR, wysłanym przez mechanizm integracyjny GitHub Discord MyRepoBot. Źródło: Opracowanie własne.	11
Rysunek 4 Główny widok użytkownika w wersji demonstracyjnej otwartoźródłowego oprogramowania Snipe-IT. Źródło: Opracowanie własne.	15
Rysunek 5 Widok zarządzania niestandardowymi polami w narzędziu Snipe IT. Źródło: Opracowanie własne.	16
Rysunek 6 Główny widok aplikacji GLPI, przedstawiający podsumowanie wszystkich encji w systemie. Źródło: Opracowanie własne.	17
Rysunek 7 Główny kokpit administracyjny EZO Inventory. Źródło: Opracowanie własne. ..	19
Rysunek 8 Widok zarządzania środkami trwałymi w aplikacji Asset Panda. Źródło: [7].....	21
Rysunek 9 Diagram aktorów systemu Glowing Waffle AMS. Źródło: Opracowanie własne.	34
Rysunek 10 Diagram przypadków użycia dla roli Pracownik. Źródło: Opracowanie własne.	35
Rysunek 11 Diagram przypadków użycia dla roli Audytor inwentaryzacji. Źródło: Opracowanie własne.	36
Rysunek 12 Diagram przypadków użycia dla roli Inwentaryzator. Źródło: Opracowanie własne.	37
Rysunek 13 Diagram przypadków użycia dla roli Zarządca wypożyczeń. Źródło: Opracowanie własne.	38
Rysunek 14 Diagram przypadków użycia dla roli Obsługujący incydenty. Źródło: Opracowanie własne.	39
Rysunek 15 Diagram przypadków użycia dla roli Zarządca środków. Źródło: Opracowanie własne.	40
Rysunek 16 Diagram klas mikroserwisu Asset Service. Źródło: Opracowanie własne.	42
Rysunek 17 Diagram klas mikroserwisu Inventory Service. Źródło: Opracowanie własne....	43
Rysunek 18 Diagram klas mikroserwisu Loan Service. Źródło: Opracowanie własne.	44
Rysunek 19 Diagram klas mikroserwisu Incident Service. Źródło: Opracowanie własne	45
Rysunek 20 Diagram klas mikroserwisu Reporting Service. Źródło: Opracowanie własne. ..	46
Rysunek 21 Diagram stanów cyklu życia kampanii inwentaryzacyjnej. Źródło: Opracowanie własne.....	47
Rysunek 22 Diagram stanów cyklu życia wypożyczenia. Źródło: Opracowanie własne	48
Rysunek 23 Diagram aktywności procesu skanowania środka trwałego w trakcie kampanii inwentaryzacyjnej. Źródło: Opracowanie własne.	49
Rysunek 24 Diagram aktywności procesu wypożyczenia środka trwałego. Źródło: Opracowanie własne.	50
Rysunek 25 Architektura systemu Glowing Waffle AMS. Źródło: Opracowanie własne.	68
Rysunek 26 Wspólny język na diagramie Venna. Źródło: Opracowanie własne.	69
Rysunek 27 Realizacja wzorca "Database per service". Źródło: Opracowanie własne.	70
Rysunek 28 Blokowanie głównego żądania przez żądania pochodne. Źródło: Opracowanie własne.....	72

Rysunek 29 Diagram sekwencji komunikacji w stylu RPC w brokerze RabbitMQ. Źródło: Opracowanie własne.	74
Rysunek 30 Diagram sekwencji autoryzacji typu Authorization Code Flow z rozszerzeniem PKCE. Źródło: Opracowanie własne.	78
Rysunek 31 Dymek typu pop-up informujący użytkownika o konieczności przedłużenia sesji. Źródło: Opracowanie własne.	79
Rysunek 32 Realizacja wzorca Transactional Outbox na przykładzie serwisu zamówień. Źródło: [44]	81
Rysunek 33 Realizacja wzorca API Composition na przykładzie metody getOrderDetails(). Źródło: [44]	81
Rysunek 34 Algorytm AWS Signature Version 4 (SigV4), używany przez bibliotekę AWS SDK do generowania podpisów kryptograficznych dla nagłówków i żądań URL. Źródło: [47]	84
Rysunek 35 Ślad żądania pobierającego wypożyczenia w systemie Glowing Waffle AMS w stosie obserwowalności Grafana Cloud. Źródło: Opracowanie własne.	87
Rysunek 36 Wizualizacja przykładowego ułożenia N procesorów danych w ramach potoku w kolektorze OpenTelemetry. Źródło: [49]	88
Rysunek 37 Przepływ żądania utworzenia środka trwałego (POST /assets). Źródło: opracowanie własne	98
Rysunek 38 Widok Swagger UI. Źródło: opracowanie własne	104
Rysunek 39 Encja Asset wraz z asocjacjami i mapowaniami typu ORM. Źródło: Opracowanie własne.	106
Rysunek 40 Wycinek pliku konfiguracyjnego brokera RabbitMQ. Źródło: Opracowanie własne.	111
Rysunek 41 Ekran powitalny systemu Glowing Waffle AMS. Źródło: Opracowanie własne.	139
Rysunek 42 Pulpit nawigacyjny systemu Glowing Waffle AMS w widoku administratora. Źródło: Opracowanie własne.	141
Rysunek 43 Formularz tworzenia i edytowania dokumentu zakupu. Źródło: Opracowanie własne.	142
Rysunek 44 Widok szczegółów zakupu z widocznym ostrzeżeniem o rozbieżności wartości oraz tabelą powiązanych środków trwałych. Źródło: Opracowanie własne.	143
Rysunek 45 Widok listy lokalizacji prezentowany w formie listowym. Źródło: Opracowanie własne.	144
Rysunek 46 Widok szczegółowy definicji pól przedstawiający tabele powiązanych kategorii oraz typów. Źródło: Opracowanie własne.	145
Rysunek 47 Widok listy stawek VAT prezentowany w formie kafelkowej. Źródło: Opracowanie własne.	146
Rysunek 48 Widok listy środków trwałych w trybie tabeli z widocznym paskiem wyszukiwania globalnego, nagłówkami kolumn z ikonami sortowania oraz przyciskami akcji w ostatniej kolumnie. Źródło: Opracowanie własne.	146
Rysunek 49 Dwuetapowy formularz dodawania środka trwałego: pierwszy modal z polami podstawowymi z przyciskiem "Save 1/2" oraz drugi modal z polami niestandardowymi z przyciskiem "Save 2/2". Źródło: Opracowanie własne.	148
Rysunek 50 Górna sekcja widoku szczegółowego środka trwałego z widocznymi sekcjami Asset Information, Location, Assigned To oraz Custom Information. Źródło: Opracowanie własne.	149

Rysunek 51 Sekcja Identifiers widoku szczegółów środka trwałego z widocznym kodem QR, kodem kreskowym oraz przyciskami drukowania. Źródło: Opracowanie własne.....	149
Rysunek 52 Sekcja Purchase Information oraz Inventory Tracking widoku szczegółów środka trwałego z widocznymi informacjami o zamówieniu oraz ostatnio wykonanej kampanii inwentaryzacyjnej. Źródło: Opracowanie własne.	150
Rysunek 53 Sekcja Availability Calendar widoku szczegółów środka trwałego z zaznaczonymi okresami wypożyczenia na tle siatki miesięcy. Źródło: Opracowanie własne.	150
Rysunek 54 Sekcja Asset History widoku szczegółów środka trwałego z przykładowymi oznaczeniami zmian. Źródło: Opracowanie własne.....	151
Rysunek 55 Sekcja Request History widoku szczegółów środka trwałego z wypisanymi akcjami związanymi wypożyczeniami. Źródło: Opracowanie własne.	151
Rysunek 56 Widok listy kampanii inwentaryzacyjnych w trybie tabelarycznym z widocznym statusem każdej kampanii, datą rozpoczęcia i zakończenia oraz osobą tworzącą. Źródło: Opracowanie własne.	152
Rysunek 57 Widok szczegółów kampanii z widocznym paskiem zakładek, nagłówkiem z przyciskami akcji zależnymi od statusu oraz zakładką Campaign Information. Źródło: Opracowanie własne	153
Rysunek 58 Modal potwierdzenia skanowania środka w trakcie aktywnej kampanii z widocznymi danymi środka, oczekiwaną lokalizacją oraz przyciskami potwierdzenia i zgłoszenia rozbieżności. Źródło: Opracowanie własne.	154
Rysunek 59 Podwidok szczegółów rozbieżności z widocznym typem rozbieżności, oczekiwaną lokalizacją środka oraz przyciskiem powrotu do kampanii. Źródło: Opracowanie własne.....	155
Rysunek 60 Widok główny obszaru wypożyczeń z widocznym paskiem czterech zakładek, licznikami rekordów oraz tabelą wniosków o wypożyczenie. Źródło: Opracowanie własne.	156
Rysunek 61 Modal tworzenia wniosku o wypożyczenie z sekcją wyboru środków trwałych, uzasadnienia i kalendarza przyjmującego okres wypożyczenia. Źródło: Opracowanie własne.	156
Rysunek 62 Modal podejmowania decyzji o wniosku z sekcją szczegółów wniosku w nagłówku (lista środków, okres, uzasadnienie) oraz formularzem decyzji poniżej. Źródło: Opracowanie własne.	157
Rysunek 63 Zakładka Finalized Requests z widoczną tabelą wypożyczeń, kolorowymi oznaczeniami statusów oraz przyciskami Issue i Return w ostatniej kolumnie. Źródło: Opracowanie własne.	158
Rysunek 64 Widok listy incydentów w trybie tabelarycznym z widoczną kolumną priorytetu, statusu oraz przypisanego środka trwałego. Źródło: Opracowanie własne.	159
Rysunek 65 Widok szczegółów incydentu z widocznymi sekcjami Incident Information, Comments oraz Attachments. Źródło: Opracowanie własne.	160
Rysunek 66 Widok listy personelu w formie tabeli z kolumnami imienia, nazwiska, adresu e-mail i statusu konta. Źródło: Opracowanie własne.	161
Rysunek 67 Widok szczegółów użytkownika z sekcją informacyjną. Źródło: Opracowanie własne.....	161
Rysunek 68 Sekcja Active Equipment Requests widoku szczegółów użytkownika wyświetlający wypożyczane środki trwałe, datę rozpoczęcia i datę zakończenia. Źródło: Opracowanie własne.	162

Rysunek 69 Sekcja Assigned Assets widoku szczegółów użytkownika wyświetlający przypisane środki trwałe wyświetlający podstawowe informacje. Źródło: Opracowanie własne.	162
Rysunek 70 Sekcja Assigned Campaigns widoku szczegółów użytkownika wyświetlający kampanie inwentaryzacyjne w których osoba bierze udział. Źródło: Opracowanie własne..	162
Rysunek 71 Widok My Assets z tabelą środków trwałych przypisanych do zalogowanego użytkownika z widocznymi filtrami kolumnowymi kategorii i stanu technicznego. Źródło: Opracowanie własne.	163
Rysunek 72 Widok My Incidents z tabelą własnych zgłoszeń, oznaczeniami priorytetu i statusu oraz przyciskiem Report Incident. Źródło: Opracowanie własne.	164
Rysunek 73 Widok My Equipment Requests z zakładkami Equipment Requests i Finalized Requests, tabelą wniosków z oznaczeniami statusów oraz przyciskiem tworzenia nowego wniosku. Źródło: Opracowanie własne.	165
Rysunek 74 Widok My Inventory Campaigns z listą kampanii przypisanych do zalogowanego użytkownika, widocznym statusem każdej z nich oraz oznaczeniem aktywnej kampanii. Źródło: Opracowanie własne.	166
Rysunek 75 Widok ciała żądania i odpowiedzi do końcówki loan-requests w Loan Svc. Źródło: Opracowanie własne.	168
Rysunek 76 Część żądań w kolekcji Asset Service. Źródło: Opracowanie własne.	169
Rysunek 77 Wykres przedstawiający wirtualnych użytkowników, współczynnik błędów, czasów odpowiedzi na osi czasu dla końcówki POST asset. Źródło: Opracowanie własne..	173
Rysunek 78 Podsumowanie testu wydajnościowego infrastruktury backendowej systemu Glowing Waffle AMS. Źródło: Opracowanie własne.	174
Rysunek 79 Zbiór wszystkich kontenerów Docker na środowisku deweloperskim. Źródło: Opracowanie własne.	175

12. Spis tabel

Tabela 1 Rozdzielenie zakresu odpowiedzialności w pracy przy implementacji prototypu.	
Źródło: Opracowanie własne.	12
Tabela 2 Szczegółowy rozkład autorstwa rozdziałów tekstu pracy inżynierskiej. Źródło:	
Opracowanie własne.	13
Tabela 3 Wymagania funkcjonalne - zarządzanie zasobami. Źródło: Opracowanie własne. ...	26
Tabela 4 Wymagania funkcjonalne - klasyfikacja zasobów i pola niestandardowe. Źródło:	
Opracowanie własne.	27
Tabela 5 Wymagania funkcjonalne - zarządzanie lokalizacjami w strukturze organizacji.	
Źródło: Opracowanie własne.	28
Tabela 6 Wymagania funkcjonalne - finansowa ewidencja zasobów. Źródło: Opracowanie	
własne.	28
Tabela 7 Wymagania funkcjonalne - Wypożyczenia środków trwałych. Źródło: Opracowanie	
własne.	29
Tabela 8 Wymagania funkcjonalne - wypożyczenia (działania automatyczne systemu).	
Źródło: Opracowanie własne.	30
Tabela 9 Wymagania funkcjonalne - kampanie inwentaryzacyjne. Źródło: Opracowanie	
własne.	30
Tabela 10 Wymagania funkcjonalne – kampanie inwentaryzacyjne (działania automatyczne	
systemu). Źródło: opracowanie własne.	31
Tabela 11 Wymagania funkcjonalne - obsługa incydentów. Źródło: Opracowanie własne.	32
Tabela 12 SLO oraz SLI systemu Głowing Waffle AMS. Źródło: Opracowanie własne.	41
Tabela 13 Scenariusz przepływu dodania zasobu. Źródło: Opracowanie własne.	51
Tabela 14 Scenariusz przepływu złożenia wniosku o wypożyczenie. Źródło: Opracowanie	
własne.	51
Tabela 15 Scenariusz przepływu rozpatrzenia wniosku o wypożyczenie. Źródło: Opracowanie	
własne.	52
Tabela 16 Domyślne ścieżki przekierowania po zalogowaniu dla poszczególnych ról. Źródło:	
Opracowanie własne.	140
Tabela 17 Kafelki statystyczne pulpitu nawigacyjnego i uprawnione do nich role. Źródło:	
Opracowanie własne.	140
Tabela 18 Akcje dostępne w widoku szczegółów kampanii w zależności od statusu. Źródło:	
Opracowanie własne.	152
Tabela 19 Statusy incydentów i ich znaczenie. Źródło: Opracowanie własne.	159
Tabela 20 Zbiorcza ocena spełnienia wymagań funkcjonalnych w podziale na obszary	
specyfikacji z podrozdziału 3.5. Źródło: Opracowanie własne.	171
Tabela 21 Rozkład czasu odpowiedzi końcówek reprezentatywnych przy obciążeniu 100 VU	
(strefa AWS Frankfurt, czas trwania 4 min, 21700 żądań na końcówkę). Wartości w	
milisekundach. Źródło: Opracowanie własne.	173

13. Spis listingów

Listing 1 Przykład pozytywnej odpowiedzi od końcówki typu <i>presign</i> , z URL potrzebnym do przesłania pliku docelowego. URL opatrzony jest wieloma parametrami służącymi do autoryzacji żądania. Źródło: Opracowanie własne.	83
Listing 2 Przykład pozytywnej odpowiedzi od końcówki typu <i>presign</i> , z URL potrzebnym do przesłania pliku docelowego. URL opatrzony jest wieloma parametrami służącymi do autoryzacji żądania. Źródło: Opracowanie własne.	83
Listing 3 Konfiguracja pipeline'ów OpenTelemetry Collector w prototypie. Wariant produkcyjny z użyciem stosu Grafana Cloud observability. Źródło: Opracowanie własne.	89
Listing 4 Implementacja wzorca Specyfikacji. Źródło: Opracowanie własne.	92
Listing 5 Klasa KeycloakRoleConverter realizująca wzorzec adaptera. Źródło: Opracowanie własne.	93
Listing 6 Komponent SecurityComponent jako fasada kontekstu bezpieczeństwa. Źródło: Opracowanie własne.	94
Listing 7 Realizacja wzorca obserwatora na przykładzie AssetHistoryPublisher. Źródło: Opracowanie własne.	95
Listing 8 Plik POM (Project Object Model) nadrzędnego projektu prototypu. Źródło: Opracowanie własne.	96
Listing 9 Struktura pakietów mikroserwisu Asset Service. Źródło: Opracowanie własne.	97
Listing 10 Skrócony fragment kontrolera AssetController – odwzorowanie metod HTTP na operacje CRUD. Źródło: Opracowanie własne.	98
Listing 11 Skrócony fragment serwisu AssetService – tworzenie środka trwałego. Źródło: opracowanie własne	99
Listing 12 Skrócony fragment serwisu AssetService - odczyt środka trwałego i odwzorowanie encji na DTO. Źródło: Opracowanie własne.	100
Listing 13 Wybrane metody repozytorium AssetRepository. Źródło: Opracowanie własne.	101
Listing 14 Uproszczony fragment encji Asset: wybrane pola i mapowanie obiektowo-relacyjne (JPA). Źródło: Opracowanie własne.	102
Listing 15 Skrócony fragment interfejsu mappera AssetMapper (MapStruct) z trzema podstawowymi odwzorowaniami. Źródło: Opracowanie własne.	102
Listing 16 Fragment obiektu żądania AssetCreateRequest z regułami walidacji. Źródło: Opracowanie własne.	103
Listing 17 Skrócony fragment encji Asset z asocjacjami oraz atrybutami niestandardowymi zapisywanymi jako JSONB. Źródło: Opracowanie własne.	104
Listing 18 Metoda przedstawiająca działanie wzorca Specyfikacji i wyszukiwania po atrybutach niestandardowych przy pomocy specjalnej funkcji PostgreSQL. Źródło: Opracowanie własne.	106
Listing 19 Walidator pojedynczego pola: adnotacja @UpperCase wraz z walidatorem. Źródło: Opracowanie własne.	107
Listing 20 Walidacja międzypolowa: adnotacja klasowa @GrossNotLessThanNet, interfejs NetGrossHolder oraz walidator. Źródło: Opracowanie własne.	108
Listing 21 Skrócony fragment globalnej obsługi wyjątków (@RestControllerAdvice) z mapowaniem wyjątków na kody HTTP. Źródło: Opracowanie własne.	109
Listing 22 Dynamiczna konfiguracja Beanów typu Queue, Binding oraz Exchange w pliku konfiguracyjnym do klienta RabbitMQ. Źródło: Opracowanie własne.	112
Listing 23 Kontroler AssetRPCController. Źródło: Opracowanie własne.	113

Listing 24 Metoda zmieniająca status środka trwałego w konsekwencji otrzymanego żądania typu RPC. Źródło: Opracowanie własne.	114
Listing 25 Współdzielony kontrakt żądań AssetRequests. Źródło: Opracowanie własne.	114
Listing 26 Współdzielony kontrakt odpowiedzi AssetResponses. Źródło: Opracowanie własne.	115
Listing 27 Fragment definicji zmiennych kolorystycznych w pliku globals.css. Źródło: Opracowanie własne.	116
Listing 28 Fragment implementacji komponentu Button z użyciem biblioteki class-variance-authority. Źródło: Opracowanie własne.	116
Listing 29 Implementacja funkcji cn łączącej biblioteki clsx i tailwind-merge. Źródło: Opracowanie własne.	117
Listing 30 Uprozczone drzewo katalogów projektu klienckiego. Źródło: Opracowanie własne.	117
Listing 31 Implementacja komponentu SessionGuard chroniącego trasy przed nieuwierzytelnionym dostępem. Źródło: Opracowanie własne.	118
Listing 32 Definicja typu Asset wraz z enumami Condition i Currency. Źródło: Opracowanie własne.	119
Listing 33 Definicja typu AssetSearchRequest reprezentującego kryteria wyszukiwania środków trwałych. Źródło: Opracowanie własne.	120
Listing 34 Fragment pliku apiCalls.ts dla mikroserwisu środków trwałych ilustrujący wzorzec wywołań CRUD przed funkcją apiClient. Źródło: Opracowanie własne.	120
Listing 35 Implementacja Route Handlera odświeżającego token dostępu przez końcówkę Keycloak. Źródło: Opracowanie własne.	121
Listing 36 Fragment interfejsu AssetContextType definiującego kontrakt kontekstu zasobów. Źródło: Opracowanie własne.	122
Listing 37 Implementacja mechanizmu leniwego ładowania w AssetContext. Źródło: Opracowanie własne.	123
Listing 38 Inicjalizacja uniwersalnej tabeli przy użyciu hooka useReactTable. Źródło: Opracowanie własne.	123
Listing 39 Fragment implementacji komponentu TableFilters z mechanizmem debounce i przełączaniem widoczności kolumn. Źródło: Opracowanie własne.	124
Listing 40 Konfiguracja callbacków jwt i session biblioteki next-auth. Źródło: Opracowanie własne.	125
Listing 41 Implementacja zdarzenia signOut propagującego wylogowanie do serwera Keycloak. Źródło: Opracowanie własne.	126
Listing 42 Katalog ról systemu w klasie AmsRole modułu ams-shared-api. Źródło: Opracowanie własne.	126
Listing 43 Fragment konfiguracji Spring Security Asset Service z podziałem uprawnień według metod HTTP. Źródło: Opracowanie własne.	127
Listing 44 Implementacja KeycloakRoleConverter mapującego role Keycloak na uprawnienia Spring Security. Źródło: Opracowanie własne.	128
Listing 45 Fragment implementacji komponentu SecurityComponent wyodrębniający dane użytkownika z tokenu JWT. Źródło: Opracowanie własne.	128
Listing 46 Zastosowanie adnotacji @PreAuthorize z weryfikacją własności rekordu w kontrolerze wniosków o wypożyczenie. Źródło: Opracowanie własne.	129
Listing 47 Konfiguracja polityki CORS z rozróżnieniem środowiska deweloperskiego i produkcyjnego. Źródło: Opracowanie własne.	130

Listing 48 Implementacja funkcji <code>isTokenValid</code> wykonującej introspekcję tokenu przez końcówkę Keycloak. Źródło: Opracowanie własne.	130
Listing 49 Warunkowe renderowanie elementów interfejsu w widoku szczegółów środka trwałego na poziomie ról użytkownika. Źródło: Opracowanie własne.	131
Listing 50 Wieloetapowy proces wytwarzania obrazu kontenera Asset Service w Dockerfile. Źródło: Opracowanie własne.	132
Listing 51 Wieloetapowy proces wytwarzania obrazu kontenera Frontend Service w Dockerfile. Źródło: Opracowanie własne.	133
Listing 52 Skrócona definicja mikroserwisu oraz jego bazy danych w pliku <code>docker-compose.prod.yaml</code> . Źródło: Opracowanie własne.	134
Listing 53 Segmenty wirtualnej sieci kontenerowej w środowisku produkcyjnym. Źródło: Opracowanie własne.	135
Listing 54 Skrócony przepływ budowy i publikacji obrazów kontenerowych w Github Container Registry (GHCR). Źródło: Opracowanie własne.	136
Listing 55 Skrócony przepływ wdrożeniowy środowiska produkcyjnego. Źródło: Opracowanie własne.	137
Listing 56 Test jednostkowy weryfikujący poprawność rzucanego wyjątku. Źródło: Opracowanie własne.	170