Dokumentacja backendu oraz bazdy danych

Łukasz Cudo oraz Kajetan Spychała

Spis treści

1 Wstęp

W poniższym dokumencie zostały zawarte szczegóły funkcjonalności projektowanej aplikacji oraz opis aspektów technicznych dotyczących backendu oraz bazy danych. Każde z zamieszczonych wymagań funkcjonalnych został wykonany i zaimplementowany w projekcie. Aplikacja spełnia wszystkie założenia dotyczące obsługi serwisu komputrowego umożliwiającego zarządzanie klientami, zleceniami napraw, zamówieniami części, magazynem części zamiennych oraz generowaniem raportów.

2 Zakres projektu

Zakres projektu obejmuje:

- implementację backendu w języku Java
- zaprojektowanie i utworzenie relacyjnej bazy danych(PostgreSQL)
- przygotowanie migracji schematu bazy danych(Flyway)
- implementację logiki biznesowej
- udostępnienie REST API
- dokumentację API przy użyciu Swaggera
- uruchamianie systemu przy pomocy Dockera

3 Wymagania biznesowe

System powinien:

- umożliwiać rejestrację i przegląd klientów serwisu,
- umożliwiać tworzenie zleceń napraw wraz z opisem usterki i datą przyjęcia,
- umożliwiać przypisywanie pracowników do zleceń,
- umożliwiać rejestrację zamówień powiązanych ze zleceniami,
- umożliwiać zmianę statusów realizacji napraw,
- umożliwiać prowadzenie magazynu części zamiennych,
- umożliwiać generowanie prostych raportów.

ID	Wymaganie	Lokalizacja w Swaggerze
WF.01	Dodawanie klienta	klient-controller `POST /api/klienci`
WF.02	Edycja danych klienta	klient-controller `PUT /api/klienci/{id}`
WF.03	Utworzenie zlecenia naprawy	zlecenie-controller `POST /api/zlecenia`
WF.04	Zmiana statusu zlecenia	zlecenie-controller `PATCH /api/zlecenia/{id}/status`

WF.05	Zarządzanie zamówieniami	zamowienie-controller
		`POST /api/zamowienia`
		`GET /api/zamowienia`
		`PATCH /api/zamowienia/{id}/odebrane`

WF.06	Zarządzanie magazynem części	czesc-controller
		`POST /api/czesci`
		`GET /api/czesci`
		`PATCH /api/czesci/{id}/stan`
		`GET /api/czesci/{id}/audit`

WF.07	Generowanie raportów	raport-controller
		`GET /api/raporty/1`
		`GET /api/raporty/2`
		`GET /api/raporty/3`
		`GET /api/raporty/1/export`

Tabela 1: Wymagania funkcjonalne systemu i ich realizacja w API

4 Wymagania niefunkcjonalne

System spełnia podstawowe wymagania niefunkcjonalne w zakresie wydajności, bezpieczeństwa oraz przenośności. Zastosowanie Spring Boot oraz Dockera umożliwia uruchomienie aplikacji w różnych środowiskach oraz jej dalszą rozbudowę.

5 Architektura systemu

System składa się z backendu napisanego w Spring Boot oraz relacyjnej bazy danych PostgreSQL. Komunikacja pomiędzy komponentami odbywa się poprzez REST API.

Warstwa	Technologia
Backend	Java 17, Spring Boot
ORM	Spring Data JPA
Baza danych	PostgreSQL
Migracje	Flyway
Dokumentacja API	Swagger / OpenAPI
Kontenery	Docker, Docker Compose

Tabela 2: Technologie użyte w aplikacji

6 Struktura bazy danych

Baza danych systemu **Serwis Komputerowy** została zaprojektowana w relacyjnym modelu danych i zaimplementowana w systemie *PostgreSQL* zgodnie z dokumentacją z etapu 2 oraz jej diagramem ERD. Jedynym odstępkiem od poprzednich ustaleń jest dodanie dwóch nowych tabel dotyczących części, tak aby ułatwić sobie implementacje backendu i prace na zbiorach danych.

6.1 Tabela `klienci`

Tabela przechowuje dane klientów korzystających z usług serwisu. Każdy klient może być powiązany z wieloma zleceniami napraw.

- `id_klienta` - unikalny identyfikator klienta (klucz główny)
- `imie` - imię klienta
- `nazwisko` - nazwisko klienta
- `nr_telefonu` - numer telefonu kontaktowego
- `adres` - adres klienta

6.2 Tabela `pracownik`

Tabela zawiera informacje o pracownikach serwisu odpowiedzialnych za realizację napraw.

- `id_pracownika` - identyfikator pracownika
- `imie` - imię pracownika

6.3 Tabela `status`

Tabela słownikowa przechowująca możliwe statusy zleceń napraw.

- `id_statusu` - identyfikator statusu
- `status` - nazwa statusu (np. Przyjęte, W realizacji)

6.4 Tabela `urzadzenie`

Tabela słownikowa przechowująca typy urządzeń obsługiwanych przez serwis.

- `id_urzadzenia` - identyfikator urządzenia
- `urzadzenie` - nazwa urządzenia

6.5 Tabela `zlecenia`

Tabela centralna systemu przechowująca informacje o zleceniach napraw.

- `id_zlecenia` - identyfikator zlecenia
- `klient` - identyfikator klienta
- `urzadzenie` - typ urządzenia
- `model_urzadzenia` - model urządzenia
- `opis_usterki` - opis zgłoszonej usterki
- `data` - data przyjęcia zlecenia
- `status` - aktualny status zlecenia
- `postep_naprawy` - opis postępu naprawy
- `pracownik` - przypisany pracownik
- `version` - pole wersjonowania rekordu

6.6 Tabela do_zamowienia

Tabela przechowuje listę elementów możliwych do zamówienia w ramach realizacji zleceń.

- id_zamowionego_przedmiotu - identyfikator przedmiotu
- zamowiony_przedmiot - nazwa przedmiotu

6.7 Tabela zamowienia

Tabela zawiera informacje o zamówieniach powiązanych z konkretnymi zleceniami napraw.

- id_zamowienia - identyfikator zamówienia
- zlecenie - identyfikator zlecenia
- zamowiony_przedmiot - zamówiony przedmiot
- cena - koszt zamówienia
- odebrane - informacja o odbiorze zamówienia

6.8 Tabela czesci_zamienne

Tabela magazynowa przechowująca informacje o częściach zamiennych.

- id_czesci - identyfikator części
- nazwa - nazwa części
- nr_katalogowy - numer katalogowy
- ilosc - ilość dostępnych sztuk
- lokalizacja - lokalizacja w magazynie
- created_at - data utworzenia rekordu
- updated_at - data ostatniej modyfikacji
- version - wersjonowanie rekordu

6.9 Tabela czesc_audit

Tabela audytowa rejestrująca zmiany ilości części zamiennych w magazynie.

- id_audytu - identyfikator wpisu audytu
- czesc_id - identyfikator części
- zmiana - wartość zmiany ilości
- powod - przyczyna zmiany
- utworzono - data wykonania operacji

7 Relacje pomiędzy tabelami

Relacje pomiędzy tabelami w bazie danych zostały zaprojektowane w taki sposób, aby odzwierciedlały rzeczywiste procesy zachodzące w serwisie komputerowym. Zastosowano klucze obce, które zapewniają spójność referencyjną danych oraz poprawność logiczną modelu.

7.1 Relacja: klienci – zlecenia

Relacja pomiędzy tabelami `klienci` oraz `zlecenia` jest relacją typu **jeden do wielu (1:N)**.

- Jeden klient może posiadać wiele zleceń napraw.
- Każde zlecenie naprawy jest przypisane do dokładnie jednego klienta.

Relacja realizowana jest poprzez klucz obcy:

- `zlecenia.klient` → `klienci.id_klienta`

7.2 Relacja: urzadzenie – zlecenia

Relacja pomiędzy tabelami `urzadzenie` oraz `zlecenia` jest relacją **jeden do wielu (1:N)**.

- Jeden typ urządzenia może występować w wielu zleceniach.
- Każde zlecenie dotyczy jednego rodzaju urządzenia.

Klucz obcy:

- `zlecenia.urzadzenie` → `urzadzenie.id_urzadzenia`

7.3 Relacja: status – zlecenia

Relacja pomiędzy tabelami `status` oraz `zlecenia` ma charakter **jeden do wielu (1:N)**.

- Jeden status może być przypisany do wielu zleceń.
- Każde zlecenie posiada dokładnie jeden aktualny status.

Klucz obcy:

- `zlecenia.status` → `status.id_statusu`

7.4 Relacja: pracownik – zlecenia

Relacja pomiędzy tabelami `pracownik` oraz `zlecenia` jest relacją **jeden do wielu (1:N)** i ma charakter opcjonalny.

- Jeden pracownik może realizować wiele zleceń.
- Zlecenie może, ale nie musi, mieć przypisanego pracownika.

Klucz obcy:

- `zlecenia.pracownik` → `pracownik.id_pracownika`

7.5 Relacja: zlecenia – zamowienia

Relacja pomiędzy tabelami `zlecenia` oraz `zamowienia` jest relacją **jeden do wielu (1:N)**.

- Jedno zlecenie może posiadać wiele zamówień.
- Każde zamówienie jest powiązane z jednym zleceniem.

Relacja posiada regułę usuwania kaskadowego:

- Usunięcie zlecenia powoduje usunięcie powiązanych zamówień.

Klucz obcy:

- `zamowienia.zlecenie` → `zlecenia.id_zlecenia`

7.6 Relacja: do_zamowienia – zamowienia

Relacja pomiędzy tabelami `do_zamowienia` oraz `zamowienia` jest relacją **jeden do wielu (1:N)**.

- Jeden element możliwy do zamówienia może występować w wielu zamówieniach.
- Każde zamówienie dotyczy jednego konkretnego przedmiotu.

Klucz obcy:

- `zamowienia.zamowiony_przedmiot` → `do_zamowienia.id_zamowionego_przedmiotu`

7.7 Relacja: czesci_zamienne – czesc_audit

Relacja pomiędzy tabelami `czesci_zamienne` oraz `czesc_audit` jest relacją **jeden do wielu (1:N)**.

- Jedna część zamienna może posiadać wiele wpisów audytowych.
- Każdy wpis audytu dotyczy jednej konkretnej części.

Relacja ta umożliwia śledzenie historii zmian magazynowych.
Klucz obcy:

- `czesc_audit.czesc_id` → `czesci_zamienne.id_czesci`

8 Architektura backendu

Backend systemu serwisu komputerowego został zaimplementowany jako aplikacja **REST API** oparta o framework **Spring Boot**. Architektura aplikacji została zaprojektowana zgodnie z zasadami separacji odpowiedzialności oraz wielowarstwowego modelu logicznego.

8.1 Ogólny model architektoniczny

Aplikacja backendowa została podzielona na następujące warstwy:

- warstwa prezentacji (Controller),
- warstwa logiki biznesowej (Service),
- warstwa dostępu do danych (Repository),
- warstwa persystencji (JPA + PostgreSQL).

Każda warstwa posiada jasno określoną odpowiedzialność i komunikuje się wyłącznie z warstwą bezpośrednio poniżej.

8.2 Warstwa prezentacji – Controllers

Warstwa prezentacji została zrealizowana przy użyciu adnotacji `@RestController`. Jej zadaniem jest:

- obsługa żądań HTTP (GET, POST, PUT),
- mapowanie endpointów REST,
- walidacja danych wejściowych,
- zwracanie odpowiedzi w formacie JSON.

Kontrolery nie zawierają logiki biznesowej, a jedynie delegują zadania do warstwy serwisów.
Przykładowe kontrolery:

- `KlientController`,
- `ZlecenieController`,
- `ZamowienieController`,
- `CzescController`,
- `RaportController`.

8.3 Warstwa logiki biznesowej – Services

Warstwa serwisów odpowiada za realizację logiki biznesowej systemu. Klasy serwisowe oznaczone są adnotacją `@Service`.

Do ich głównych zadań należą:

- przetwarzanie danych przekazywanych z kontrolerów,
- realizacja przypadków użycia systemu,
- zarządzanie transakcjami,
- koordynacja operacji na wielu repozytoriach.

Metody serwisów są objęte transakcjami przy użyciu adnotacji `@Transactional`, co zapewnia spójność danych.

Przykładowe serwisy:

- `KlientService`,
- `ZlecenieService`,
- `ZamowienieService`,
- `CzescService`,
- `RaportService`.

8.4 Warstwa dostępu do danych – Repositories

Warstwa repozytoriów odpowiada za komunikację z bazą danych. Została zaimplementowana z wykorzystaniem **Spring Data JPA** oraz **EntityManager**.

Repozytoria umożliwiają:

- wykonywanie operacji CRUD,
- realizację zapytań natywnych SQL,
- mapowanie wyników zapytań na obiekty domenowe lub struktury DTO.

W systemie zastosowano dwa podejścia:

- klasyczne repozytoria JPA dla encji,
- dedykowane repozytorium raportowe wykorzystujące zapytania natywne.

Przykład:

- `RaportRepository` - realizujący złożone zapytania raportowe.

8.5 Warstwa persystencji – JPA i encje

Warstwa persystencji oparta jest o specyfikację **Jakarta Persistence (JPA)**. Każda tabela w bazie danych posiada odpowiadającą jej encję oznaczoną adnotacją `@Entity`.

Encje:

- mapują kolumny tabel na pola klas,
- definiują relacje pomiędzy tabelami (`@ManyToOne`, `@OneToMany`),
- zawierają adnotacje walidacyjne.

Do generowania metod dostępowych wykorzystano bibliotekę **Lombok**.

8.6 Migracje bazy danych

Zarządzanie strukturą bazy danych zostało zrealizowane przy użyciu narzędzia **Flyway**. Migracje SQL przechowywane są w katalogu `db/migration`.

Zastosowanie migracji zapewnia:

- automatyczne tworzenie struktury bazy danych,
- wersjonowanie schematu,
- powtarzalność środowisk.

8.7 Dokumentacja API – Swagger

Dokumentacja REST API została wygenerowana automatycznie przy użyciu **Swagger (OpenAPI)**.

Swagger umożliwia:

- przegląd dostępnych endpointów,
- testowanie operacji bezpośrednio z przeglądarki,
- wgląd w strukturę danych wejściowych i wyjściowych.

8.8 Konteneryzacja

Aplikacja backendowa oraz baza danych działają w środowisku **Docker**. Użyto pliku `docker-compose.yml`, który definiuje:

- kontener aplikacji Spring Boot,
- kontener bazy danych PostgreSQL,
- zależności pomiędzy usługami.

Dzięki konteneryzacji możliwe jest szybkie uruchomienie systemu w dowolnym środowisku.

9 Walidacja danych i obsługa błędów

W projekcie zastosowano mechanizmy walidacji danych wejściowych w oparciu o specyfikację **Jakarta Validation**. Walidacja realizowana jest na poziomie encji oraz obiektów przekazywanych do warstwy serwisowej.

Wykorzystane adnotacje walidacyjne obejmują m.in.:

- `@NotNull` - wymuszenie obecności wartości,
- `@NotBlank` - walidację pól tekstowych,
- `@Size` - ograniczenie długości pól tekstowych.

Dzięki walidacji system zapobiega zapisywaniu niepoprawnych lub niekompletnych danych do bazy danych, co zwiększa spójność i integralność informacji.

Obsługa błędów realizowana jest przez mechanizmy frameworka Spring Boot. W przypadku wystąpienia błędów walidacji lub problemów z przetwarzaniem żądań, aplikacja zwraca odpowiednie kody HTTP, takie jak:

- `400 Bad Request` - dla błędnych danych wejściowych,
- `404 Not Found` - dla nieistniejących zasobów,
- `500 Internal Server Error` - dla błędów serwera.

Informacje o błędach przekazywane są w formacie JSON, co umożliwia ich czytelną interpretację przez klienta API oraz narzędzia takie jak Swagger.

10 Podsumowanie

W ramach realizacji projektu opracowano kompletny backend systemu informatycznego wspierającego obsługę serwisu komputerowego. Zaimplementowano warstwę aplikacyjną w technologii Spring Boot oraz relacyjną bazę danych PostgreSQL, której strukturę zaprojektowano zgodnie z wymaganiami biznesowymi systemu.

System umożliwia zarządzanie klientami, zleceniami napraw, zamówieniami oraz magazynem części zamiennych. Zaimplementowano mechanizmy walidacji danych wejściowych, obsługę błędów oraz raporty wspierające analizę pracy serwisu. Komunikacja z systemem realizowana jest poprzez interfejs REST, a dostępne endpointy zostały udokumentowane przy użyciu narzędzia Swagger.

Zastosowanie migracji Flyway oraz konteneryzacji z wykorzystaniem Dockera zapewnia spójność środowiska uruchomieniowego i ułatwia dalszy rozwój systemu. Opracowane rozwiązanie spełnia założone wymagania funkcjonalne i niefunkcjonalne oraz stanowi solidną podstawę do dalszej rozbudowy aplikacji.