



Politechnika Łódzka

Instytut Informatyki

RAPORT Z PROJEKTU KOŃCOWEGO

STUDIÓW PODYPLOMOWYCH

NOWOCZESNE APLIKACJE BIZNESOWE JAVA EE

**SYSTEM REZERWACJI WIZYT W KLINICE MEDYCYNY
SPORTOWEJ**

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej

Opiekun: dr inż. Michał Karbowańczyk

Słuchacz: Katarzyna Brewczyńska

Łódź, 6 grudnia 2019 r.



Instytut Informatyki

90-924 Łódź, ul. Wólczańska 215, **budynek B9**

tel. 042 631 27 97, 042 632 97 57, fax 042 630 34 14 email: office@ics.p.lodz.pl

Spis treści

1	Cel i zakres projektu.....	3
2	Założenia projektu.....	4
2.1	Wersje zastosowanych technologii i narzędzi.....	4
2.2	Wymagania funkcjonalne.....	4
2.3	Wymagania niefunkcjonalne.....	6
2.4	Przypadki użycia dla różnych poziomów dostępu.....	7
2.4.1	Diagramy przypadków użycia.....	7
2.4.2	Tabela krzyżowa ról i przypadków użycia.....	8
2.4.3	Opis przypadków użycia.....	10
3	Realizacja projektu.....	21
3.1	Realizacja przykładowego CRUD'a.....	21
3.1.1	Tworzenie.....	21
3.1.2	Wyświetlanie.....	25
3.1.3	Edycja.....	28
3.1.4	Usuwanie.....	31
3.2	Warstwa składowania danych.....	33
3.2.1	Model relacyjnej bazy danych.....	34
3.2.2	Konfiguracja zasobów w relacyjnej bazie danych.....	34
3.2.3	Mapowanie obiektowo-relacyjne ORM.....	35
3.3	Warstwa logiki biznesowej.....	37
3.3.1	Komponenty EJB.....	38
3.3.2	Mechanizmy ochrony spójności danych.....	40
3.3.3	Kontrola dostępu.....	42
3.3.4	Kontrola odpowiedzialności.....	42
3.3.5	Obsługa błędów.....	43
3.4	Warstwa widoku.....	47
3.4.1	Wzorzec projektowy DTO.....	47
3.4.2	Ujednolicony interfejs użytkownika.....	48
3.4.3	Internacjonalizacja.....	51
3.4.4	Uwierzytelnianie i autoryzacja.....	52
3.4.5	Walidacja danych.....	54
3.4.6	Obsługa błędów.....	55
3.4.7	Technologie do obsługi interfejsu graficznego.....	56
3.5	Instrukcja wdrożenia.....	58
3.5.1	Utworzenie bazy danych.....	58
3.5.2	Umieszczenie w bazie struktur bazy danych i danych inicjujących.....	59
3.5.3	Nawiązanie połączenia z bazą danych.....	59
3.5.4	Konfiguracja obszaru bezpieczeństwa na serwerze.....	59
3.5.5	Wgranie aplikacji na serwer aplikacyjny i uruchomienie aplikacji.....	60
3.6	Podsumowanie.....	62
4	Źródła.....	64
5	Spis załączników.....	65

1 Cel i zakres projektu

Temat pracy został wybrany spośród tematów zaproponowanych przez firmę Rossmann w ramach współpracy z Politechniką Łódzką. Celem projektu jest stworzenie systemu informatycznego wspomagającego rezerwację wizyt w Klinice Medycyny Sportowej. Budowana aplikacja umożliwi pacjentom samodzielne rezerwowanie wizyt u lekarzy specjalistów na dany dzień oraz godzinę, a także usprawni pracę recepcji kliniki poprzez zarządzanie rezerwacjami.

System będzie składał się z aplikacji internetowej wykonanej w technologii Java Enterprise Edition dostępną z poziomu przeglądarki internetowej, osadzoną na serwerze aplikacyjnym Payara oraz relacyjnej bazy danych. Trójwarstwowy model umożliwi wyodrębnienie warstwy widoku, który jest dostępny z poziomu przeglądarki internetowej, warstwy logiki biznesowej oraz warstwy składowania danych w relacyjnej bazie danych. Środowisko programistyczne wykorzystane do stworzenia systemu w języku programowania Java, to NetBeans IDE.

System informatyczny będzie zapewniał funkcjonalności zależnie od poziomu dostępu, który przypisany będzie do danego użytkownika. Obsługa programu będzie dostępna w dwóch wersjach językowych: polskiej i angielskiej. Użytkownik będzie mógł zmieniać preferencje językowe w opcjach przeglądarki internetowej. Przetwarzanie transakcyjne oraz wykorzystanie blokad optymistycznych zapewni dostęp do danych wielu użytkownikom, którzy się uwierzytelnią.

Zakres projektu obejmuje:

- analizę wymagań biznesowych oraz opracowanie funkcjonalności budowanego systemu,
- wybór technologii odpowiednich do wykonania aplikacji,
- opracowanie architektury tworzonego systemu,
- opis przypadków użycia zobrazowanych diagramami,
- zaprojektowanie modelu bazy danych,
- implementacja aplikacji zsynchronizowanej z relacyjną bazą danych,
- stworzenie instrukcji wdrożenia systemu,
- przygotowanie danych inicjujących niezbędnych do uruchomienia systemu,
- próbne zdrożenie systemu na podstawie instrukcji wdrożenia.

2 Założenia projektu

Rozdział zawiera funkcjonalne oraz technologiczne założenia projektu. Szczegółowe założenia zostały zagregowane w formie podrozdziałów.

2.1 Wersje zastosowanych technologii i narzędzi

Zastosowane technologie i narzędzia podczas realizacji projektu:

Zastosowane technologie i narzędzia podczas realizacji projektu:

- Środowisko programistyczne NetBeans IDE w wersji 8.2;
- Język programowania Java 8;
- JDK wersji 1.8 update 201;
- Maven w wersji 3.6.0 używane do budowania i zarządzania projektem;
- Serwer aplikacyjny to Glassfish w wersji 4.1.;
- Java Enterprise Edition 7.0 - platforma programistyczna umożliwiająca tworzenie aplikacji w architekturze kontener-komponent w której zrealizowano logikę biznesową;
- Context and Dependency Injection w wersji 1.2;
- Java Server Faces w wersji 2.2 obsługuje warstwę widoku. Dodatkowo w aplikacji wykorzystano elementy publicznej biblioteki Java PrimeFaces w wersji 7.0 oraz Bootstrap 3.3.7.
- PrimeFaces w wersji 5.0;
- Java DB (Apache Derby) odpowiedzialna za składowanie danych w wersji 10.13.1.1;
- Enterprise Java Beans 3.2;
- HTML 5 oraz CSS 3;
- Bootstrap w wersji 3.3;
- JQuery w wersji 3.2.1;
- Java Transaction w wersji API 1.2;
- JavaScript w wersji 1.8;
- obiekty mapowane są z wykorzystaniem standardu Java Persistence API w wersji 2.1;
- diagramy wykonane zostały w Modelio Open Source 3.7.

2.2 Wymagania funkcjonalne

Realizowany projekt będzie posiadał cztery poziomy dostępu. W założeniach przyjęto, że każdy użytkownik systemu będzie musiał dysponować indywidualnym, autoryzowanym przez administratora kontem, które może mieć przypisany co najmniej jeden poziom dostępu. Każdemu z poziomów zostały przypisane inne zbiory przypadków użycia. Użytkownik nieuwierzytelniony będzie posiadał status „Gościa”, miał możliwość utworzenia i zalogowania się na własne konto. Będzie mógł również zresetować przypisane do własnego konta hasło.

- **Administrator** – zarządza aspektami związanymi z kontami wszystkich użytkowników, autoryzuje je przypisując im poziom dostępu, może je też, aktywować,

dezaktywować i edytować ich dane. Może kasować nieautoryzowane konta i zmieniać poziom dostępu kontom nie mającym powiązań z żadnymi rezerwacjami, specjalistami i innymi kontami.

- **Recepcja** – może dodawać specjalizacje oraz specjalistów, usuwać, aktywować, dezaktywować i edytować ich dane. Zajmuje się też obsługą rezerwacji pacjentów, ma możliwość odwoływania pojedynczych oraz wszystkich rezerwacji przypisanych do danego specjalisty.
- **Pacjent** – ma możliwość utworzenia rezerwacji poprzez wybór specjalizacji, następnie wybór specjalisty oraz terminu, w jakim ma odbyć się wizyta poprzez interaktywny kalendarz. Ma dostęp do listy swoich aktualnych i przeszłych rezerwacji wizyt. Może edytować lub usuwać swoje aktualne rezerwacje.

Bez względu na wybrany do przypisania poziom dostępu, po zarejestrowaniu konta użytkownik musi udać się osobiście z dowodem tożsamości do administratora systemu w celu potwierdzenia swoich danych osobowych. Po ich weryfikacji administrator przypisuje użytkownikowi odpowiedni poziom dostępu i autoryzuje jego konto.

Ponadto system będzie posiadał następujące funkcjonalności:

1. Rezerwacja wizyty będzie dotyczyła okresu jednej godziny w dni robocze, w godzinach pracy placówki tj. między godziną 8.00 a 18.00 (ostatnia rezerwacja na godzinę 17.00).
2. System pozwoli na rezerwowanie nieograniczonej liczby wizyt u różnych specjalistów, jednak nie będzie możliwe zarezerwowanie więcej niż jednej wizyty w tym samym terminie.
3. Jeden specjalista może posiadać jedną rezerwację w danym terminie.
4. System nie pozwoli na zarezerwowanie przeszłego terminu.
5. System nie pozwoli pacjentowi zarezerwować dezaktywowanego specjalisty, aż do momentu jego ponownej aktywacji.
6. Każdy uwierzytelniony użytkownik będzie miał dostęp do informacji o swoim loginie i przypisanym poziomie dostępu.
7. Każdy z uwierzytelnionych użytkowników będzie mógł wyświetlić, edytować dane, oraz zmieniać hasło swojego konta. Do przeprowadzenia zmian konieczne jest podanie swojego obecnego hasła.
8. Dezaktywowany użytkownik utraci natychmiastowo dostęp do krytycznych operacji na swoim koncie, nawet gdy jest on obecnie zalogowany. Po zakończeniu sesji nie ma możliwości ponownego zalogowania się na własne konto.

9. Rezerwacji będzie można dokonywać od dnia następnego.
10. Wizyty u specjalistów mogą być rezerwowane z maksymalnie trzymiesięcznym wyprzedzeniem.

2.3 Wymagania niefunkcjonalne

Poniżej zostały wymienione wymagania niefunkcjonalne, które będzie spełniać aplikacja.

1. System będzie zbudowany w architekturze trójwarstwowej, z podziałem na warstwę składowania danych, warstwę logiki biznesowej oraz warstwę widoku.
2. System będzie obsługiwał uwierzytelnionego użytkownika i autoryzację względem przypisanego do konta jednego poziomu dostępu.
3. Użytkownik będzie mógł korzystać ze swojego konta, jeśli posiada ono przypisany poziom dostępu, a także zostało potwierdzone przez administratora i jest aktywne.
4. Użytkownik będzie miał przydzielone uprawnienia do danych funkcji systemu w zależności od przypisanego poziomu dostępu.
5. Wzorce loginów i haseł będą przechowywane w relacyjnej bazie danych. Hasła będą przechowywane w postaci niejawnej, jako skrót wyliczony z algorytmu Secure Hash Algorithm-256.
6. System będzie wielodostępowy, co umożliwi jednoczesną pracę wielu użytkowników, a spójność danych w systemie zapewnią transakcje i blokady optymistyczne.
7. System zapewni odpowiednią obsługę błędów oraz mechanizm gromadzenia logów systemowych w dzienniku zdarzeń.
8. Interfejs użytkownika będzie zapewniony dzięki dynamicznie generowanym stronom WWW dostępnym poprzez współczesną przeglądarkę internetową.
9. Wylogowanie się uwierzytelnionego użytkownika następuje automatycznie po upływie 25 minut nieaktywności, gdy sesja uwierzytelnionego użytkownika zostanie zakończona lub na żądanie użytkownika ze skutkiem natychmiastowym,
10. System będzie posiadał internacjonalizację ograniczoną do interfejsu użytkownika.
11. System będzie posiadał polską i angielską wersję językową, jednak dane wprowadzone do aplikacji poprzez formularze nie będą podlegać internacjonalizacji.
12. System nie będzie obsługiwał znaków diakrytycznych.
13. Serwer aplikacyjny, na którym będzie działać aplikacja ma bieżący czas systemowy, który jest zgodny z aktualnym czasem.
14. System informatyczny będzie przechowywać w bazie danych informacje o ostatniej operacji na danym obiekcie wraz z datą i godziną modyfikacji tego obiektu oraz użytkownikiem, który takiej modyfikacji dokonał. Do czasu usunięcia obiektu zachowana jest również data i godzina utworzenia obiektu oraz informacja o tym, kto utworzył obiekt. Przypisane konto uwierzytelnionego użytkownika będzie

pobierane z bieżącego logowania, a data i godzina będą zbieżne z czasem systemowym.

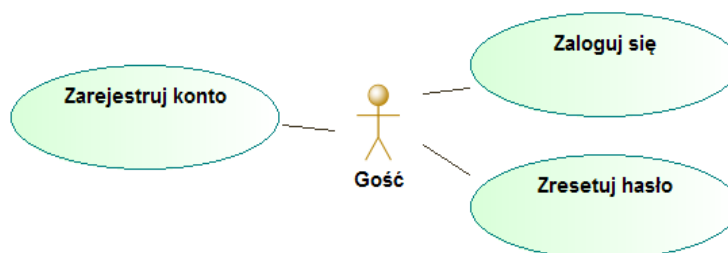
15. Aplikacja nie będzie zgodna z wymogami określonymi w Rozporządzeniu o Ochronie Danych Osobowych.

2.4 Przypadki użycia dla różnych poziomów dostępu

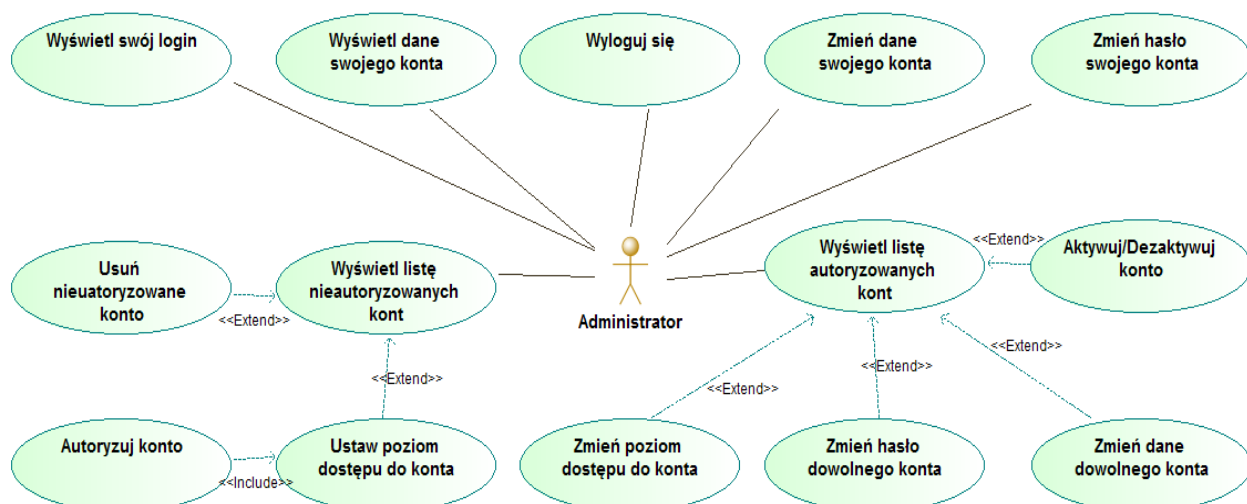
W podrozdziale 2.2 zdefiniowano cztery poziomy dostępu. Poniżej zostały one zaprezentowane w dwojaki sposób: na diagramach przypadków użycia w języku UML (ang. *Unified Modeling Language*) oraz w tabeli krzyżowej [5].

2.4.1 Diagramy przypadków użycia

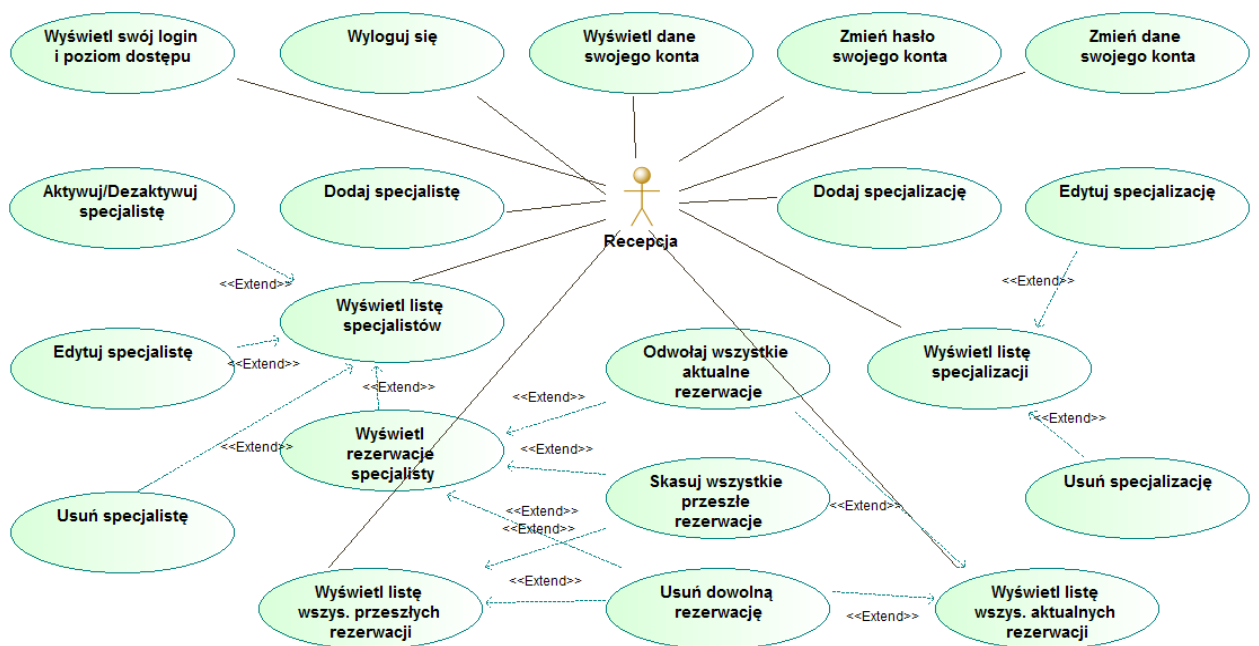
Rysunek 1 przedstawia przypadki użycia dla użytkownika nieuwierzytelnionego. Na rysunkach 2, 3, 4 przedstawione zostały przypadki użycia, dostępne dla użytkownika z przydzielonym poziomem dostępu, administrator, recepcja i pacjent.



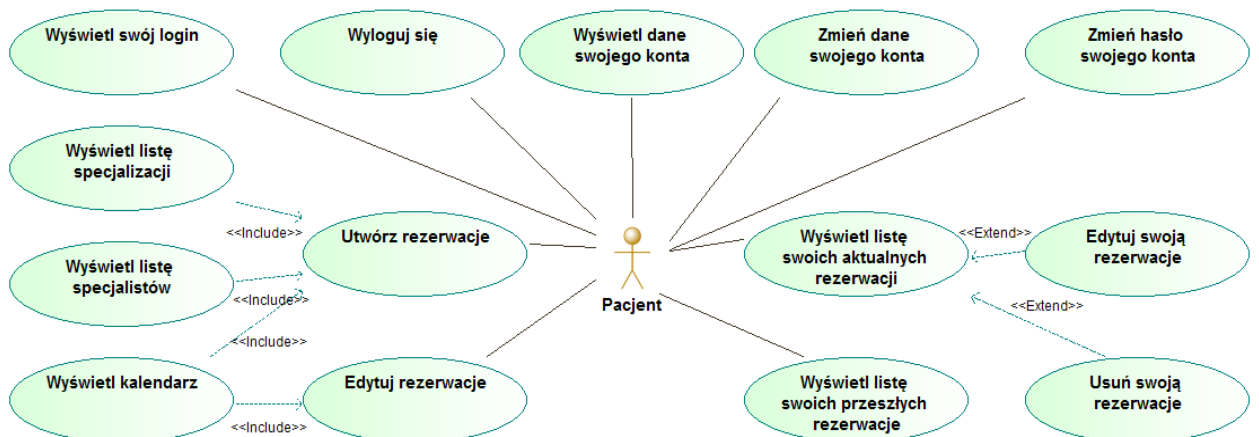
Rysunek 1: Diagram przypadków użycia dla poziomu dostępu: Gość



Rysunek 2: Diagram przypadków użycia dla poziomu dostępu: Administrator



Rysunek 3: Diagram przypadków użycia dla poziomu dostępu: Recepcja



Rysunek 4: Diagram przypadków użycia dla poziomu dostępu: Pacjent

2.4.2 Tabela krzyżowa ról i przypadków użycia

W tabeli 1 przedstawiono wszystkie przypadki użycia podzielone na grupy, zgodnie z przypisanym użytkownikowi poziomem dostępu, a także przypadki użycia użytkownika nieuwierzytelnionego.

Tabela 1. Tabela krzyżowa poziomów dostępu i przypadków użycia.

Lp.	Przypadek użycia	Gość	Administrator	Rejestracja	Pacjent
1	Zarejestruj konto	x			
2	Zaloguj się	x			
3	Zresetuj hasło	x			
4	Wyloguj się		x	x	x
5	Wyświetl dane swojego konta		x	x	x
6	Edytuj dane swojego konta		x	x	x
7	Zmień hasło swojego konta		x	x	x
8	Wyświetl swój login		x	x	x
9	Wyświetl listę nieautoryzowanych kont		x		
10	Usuń nieautoryzowane konto		x		
11	Autoryzuj konto		x		
12	Wyświetl listę autoryzowanych kont		x		
13	Zmień poziom dostępu dla konta		x		
14	Zmień dane dowolnego konta		x		
15	Aktywuj/Dezaktywuj konto		x		
16	Zmień hasło dowolnego konta		x		
17	Dodaj specjalizację			x	
18	Wyświetl listę specjalizacji			x	
19	Edytuj dane specjalizacji			x	
20	Usuń specjalizację			x	
21	Dodaj specjalistę			x	
22	Wyświetl listę specjalistów			x	
23	Edytuj dane specjalisty			x	
24	Aktywuj/Dezaktywuj specjalistę			x	
25	Usuń specjalistę			x	
26	Wyświetl listę rezerwacji specjalisty			x	
27	Wyświetl listę wszystkich aktualnych rezerwacji			x	
28	Wyświetl listę wszystkich przeszłych rezerwacji			x	
29	Utwórz rezerwację				x
30	Edytuj rezerwację				x
31	Wyświetl listę swoich aktualnych rezerwacji				x
32	Wyświetl listę swoich przeszłych rezerwacji				x
33	Usuń swoją rezerwację				x
34	Usuń dowolną rezerwację				x

Lp.	Przypadek użycia	Gość	Administrator	Rejestracja	Pacjent
35	Usuń wszystkie rezerwacje			x	
36	Usuń wszystkie przeszłe rezerwacje			x	

2.4.3 Opis przypadków użycia

W rozdziale znajduje się szczegółowy opis wszystkich przypadków użycia z załączonymi rysunkami [13].

Zarejestruj konto - przypadek użycia dostępny dla nieuwierzytelnionego użytkownika. Ma na celu wprowadzenie danych niezbędnych do rejestracji konta do formularza przedstawionego na rysunku 5. Po wybraniu przycisku „Rejestruj konto” następuje zapis danych do bazy, a konto ma przydzielany tymczasowy poziom dostępu nowo zarejestrowanego konta, które nie ma uprawnień do żadnych funkcjonalności.

Rejestracja konta

Imię

Nazwisko

Numer telefonu

Login

Hasło

Powtórz hasło

Pytanie bezpieczeństwa

Sekretna odpowiedź

Zarejestruj

Anuluj

Wszystkie pola są wymagane.

Rysunek 5: Zarejestruj konto

Zaloguj się - użytkownik podaje unikalny login oraz aktualne hasło i potwierdza przyciskiem „Zaloguj”. Oba pola są wymagane, o czym świadczy napis pod formularzem, co przedstawiono na rysunku 6. Po podaniu loginu oraz hasła, aplikacja sprawdza czy takie konto widnieje w bazie danych, oraz czy zostało autoryzowane i czy jest aktywne.

Jeśli wszystkie warunki zostaną spełnione, użytkownik uzyskuje dostęp do funkcjonalności przypisanych dla jego poziomu dostępu.

*Rysunek 6:
Zaloguj się*

Zresetuj hasło - jest to funkcjonalność dostępna dla użytkownika, który chce ustawić nowe hasło. Składa się z trzech odrębnych stron. Na pierwszej pokazanej na rysunku 7, użytkownik podaje swój login, przypisany do konta i klika „Zatwierdź”. Następnie, po sprawdzeniu w bazie danych czy podany login istnieje, następuje przekierowanie na drugą stronę, widoczną na rysunku 8, gdzie wyświetlone zostaje pytanie do resetowania hasła. Należy podać odpowiedź na pytanie, podane przez użytkownika przy rejestracji konta i nacisnąć przycisk „Zatwierdź”. Następuje kolejne przekierowanie na stronę, gdzie należy wpisać nowe hasło oraz powtórzyć wpis, celem sprawdzenia czy nie doszło do pomyłki, co widać na rysunku 9. Jeśli wszystkie dane zostały wpisane poprawnie, nowe hasło użytkownika zostaje zapisane w postaci niejawnej do bazy danych [13].

Resetowanie hasła

*Rysunek 7:
Resetowanie hasła
(Wpisz login)*

Resetowanie hasła

*Rysunek 8:
Resetowanie hasła
(Wpisz odpowiedź)*

Resetowanie hasła

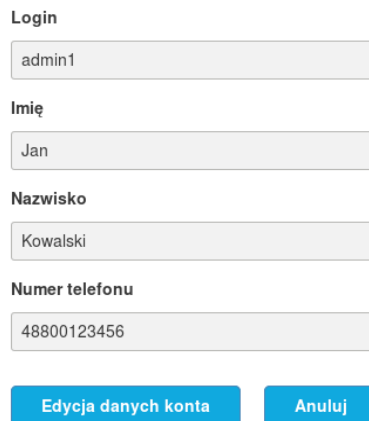
*Rysunek 9:
Resetowanie hasła
(Wpisz hasło)*

Wyloguj się - użytkownik po przyciśnięciu „Wyloguj się”, zaprezentowanego na rysunku 10 traci dostęp do funkcjonalności przypisanych dla przydzielonego mu poziomu dostępu, a zyskuje do funkcjonalności dostępnych dla użytkownika niewierzytelnionego.

Rysunek 10: Wyloguj się / Wyświetl swój login

Wyświetl dane swojego konta – umożliwia uwierzytelnionemu użytkownikowi wgląd w informacje o własnym koncie. Z tego miejsca możliwe jest edytowanie tych danych po kliknięciu na przycisk „Edycja danych konta”. Widok formularza został zaprezentowany na rysunku 11.

Wyświetl dane własnego konta



Login

admin1

Imię

Jan

Nazwisko

Kowalski

Numer telefonu

48800123456

Edycja danych konta Anuluj

Rysunek 11: Wyświetl dane własnego konta

Edytuj dane swojego konta - funkcjonalność ta daje możliwość uwierzytelnionemu użytkownikowi edycję swoich danych. Aby to zrobić konieczne jest podanie aktualnego hasła w celu weryfikacji uprawnień użytkownika. Formularz pokazuje rysunek 12.

Edycja danych własnego konta



Login

admin1

Aktualne hasło

Imię

Jan

Nazwisko

Kowalski

Numer telefonu

48800123456

Zapisz Anuluj

Rysunek 12: Edytuj dane swojego konta

Zmień hasło swojego konta – kolejny przypadek dostępny dla uwierzytelnionego użytkownika. Po podaniu poprawnego starego hasła i dwukrotnym wprowadzeniu nowego, co przedstawia rysunek 13, zapisywane jest ono w bazie danych w formie niejawnej.

Zmiana własnego hasła

Login

Stare hasło

Nowe hasło

Powtórz nowe hasło

[Zatwierdź](#) [Anuluj](#)

Rysunek 13: Zmiana własnego hasła

Wyświetl swój login – ten przypadek użycia powoduje samoczynne wywołanie metody wyświetlającej na pasku menu loginu uwierzytelnionego użytkownika, jest on widoczny na pasku menu, na rysunku 10.

Wyświetl listę nieautoryzowanych kont - umożliwia administratorowi wgląd w listę nowo zarejestrowanych kont oraz wykonanie związanych z nimi akcji, listę uwidacznia rysunek 14.

Lista nowych kont

Imię	Nazwisko	Login	Numer telefonu	Akcje
Jan	Nowak	jannowak	08003672678	<div><div>Wybierz poziom dostępu ▾</div><div>Ustaw poziom dostępu</div><div>Usuń konto</div></div>

[Powrót do strony głównej](#)

Rysunek 14: Wyświetl listę nieautoryzowanych kont

Usuń nieautoryzowane konto - powoduje usunięcie z bazy danych wybranego z listy konta. Usunięcie jest możliwe tylko dla kont jeszcze nieautoryzowanych przez administratora. Przycisk „Usuń” zaprezentowano na rysunku 14.

Autoryzuj konto - odbywa się poprzez wybranie przez administratora poziomu dostępu dla konta użytkownika wybranego z listy. Z rozwijanej listy należy zaznaczyć odpowiedni poziom dostępu, a następnie kliknąć „Ustaw poziom dostępu”, jak pokazano na rysunku 14. Po przydzieleniu poziomu dostępu konto nie jest już dłużej widoczne na liście kont nowo zarejestrowanych, a trafia na listę kont użytkowników autoryzowanych, widoczną na rysunku 15.

Wyświetl listę autoryzowanych kont - powoduje pobranie z bazy i wyświetlenie listy kont autoryzowanych użytkowników, widocznej na rysunku 15. Podobnie jak w przypadku listy kont nowo zarejestrowanych, można wybrać konto, a następnie dokonać zmian.

Lista kont

Imię	Nazwisko	Login	Numer telefonu	Akcje					
Maria	Nowak	admin	806666666	Administrator	Zmień poziom dostępu	Edytuj konto	Zmień hasło	Aktywuj	Dezaktywuj
Katarzyna	Brewczyńska	reception	801123456	Recepcja	Zmień poziom dostępu	Edytuj konto	Zmień hasło	Aktywuj	Dezaktywuj
Piotr	Graczykowski	patient	801696969	Pacjent	Zmień poziom dostępu	Edytuj konto	Zmień hasło	Aktywuj	Dezaktywuj

[Powrót do strony głównej](#)

Rysunek 15: Wyświetl listę autoryzowanych kont

Zmień poziom dostępu dla konta – umożliwia zmianę poziomu dostępu dla konta użytkownika wybranego z listy. Zmiana ta powoduje przepisanie wszystkich danych obecnego konta do nowego konta, ustawienie w nim wybranego poziomu dostępu i dodanie go do bazy danych. Obecne konto zostaje usunięte z bazy danych. Operacja ta jest możliwa wyłącznie wtedy, kiedy konto użytkownika nie zostało powiązane żadnymi relacjami w tabelach bazy danych. Listę zaprezentowano na rysunku 15.

Aktywuj/Dezaktywuj konto - dezaktywacja ma zastosowanie w przypadku gdy administrator chce zablokować użytkownikowi dostęp do aplikacji. Następnie, w razie potrzeby, można aktywować je ponownie. Na rysunku 15 przedstawiono listę kont autoryzowanych po dezaktywowaniu użytkownika. Przycisk „Dezaktywuj” pozostał wciśnięty aby nie można było ponownie wykonać tej samej akcji, zaś przycisk „Aktywuj” się uwidocznił.

Edytuj dane dowolnego konta – kiedy konieczne jest wprowadzenie zmiany w danych personalnych użytkownika przez administratora. Formularz, przedstawiony na rysunku 16, daje możliwość zmiany imienia, nazwiska i adresu. Login jest unikalny.

Edycja danych konta użytkownika

Login	<input type="text" value="admin1"/>
Imię	<input type="text" value="Jan"/>
Nazwisko	<input type="text" value="Kowalski"/>
Numer telefonu	<input type="text" value="48800123456"/>
Pytanie bezpieczeństwa	<input type="text" value="Ulubiony język programowania to?"/>
Sekretna odpowiedź	<input type="text" value="Java"/>
<div><input type="button" value="Wyślij"/> <input type="button" value="Anuluj"/></div>	

Rysunek 16: Zmień dane dowolnego konta

Zmień hasło dowolnego konta - umożliwia administratorowi zmianę hasła użytkownika, gdy przypadek zmiany własnego hasła, pokazany na rysunku 13, jak również resetowania hasła z rysunków 7, 8 i 9 nie może zostać wykonany przez samego użytkownika.

Administrator wybiera użytkownika z listy i dwukrotnie wpisuje nowe hasło, co obrazuje rysunek 17. Hasło to następnie podaje użytkownikowi, aby ten mógł wykorzystać je do zalogowania, a następnie zmienić hasło.

Zmiana hasła użytkownika

Login

admin1

Nowe hasło

Powtórz nowe hasło

Wyślij Anuluj

Rysunek 17: Zmień hasło dowolnego konta

Dodaj specjalizację – przypadek użycia dostępny dla pracowników recepcji. Po wypełnieniu formularza wymaganymi danymi i naciśnięciu przycisku „Dodaj”, dane zostają zapisane w bazie danych w odpowiednich kolumnach, formularz widoczny jest na rysunku 19.

Dodaj specjalizację

Kod specjalizacji

Nazwa specjalizacji

Dodaj Anuluj

Rysunek 19: Dodaj specjalizację

Edycja specjalizacji

Kod specjalizacji

003

Nazwa specjalizacji

Rehabilitacja

Zapisz Anuluj

Rysunek 18: Edytuj dane specjalizacji

Edytuj dane specjalizacji – umożliwia edycję nazwy specjalizacji na formularzu zaprezentowanym na rysunku 18.

Wyświetl listę specjalizacji - powoduje pobranie z bazy i wyświetlenie w formie listy wszystkich dostępnych specjalizacji. W tym miejscu możliwe jest przejście do edycji specjalizacji, a także jej usunięcie. Lista widoczna jest na rysunku 20.

Lista specjalizacji

Kod specjalizacji	Nazwa specjalizacji	Akcje	
001	Neurologia	Edytuj specjalizację	Usuń specjalizację
002	Flebologia	Edytuj specjalizację	Usuń specjalizację
003	Rehabilitacja	Edytuj specjalizację	Usuń specjalizację

Anuluj

Rysunek 20: Wyświetl listę specjalizacji

Usuń specjalizację – umożliwia pracownikowi recepcji usunięcie specjalizacji z bazy, jeżeli tylko żaden specjalista nie został jeszcze do niej przypisany. Usunięcie odbywa się poprzez wciśnięcie przycisku „Usuń specjalizację” widocznego na rysunku 20

Dodaj specjalistę – w pierwszej kolejności należy wybrać specjalizację z listy rozwijanej. Następnie konieczne jest uzupełnienie pozostałych danych i naciśnięcie przycisku „Dodaj” widocznego rysunku 23.

Dodaj specjalistę

Specjalizacja

Wybierz specjalizację: ▼

Imię specjalisty

Nazwisko specjalisty

Numer PWZ

Pusta lista rozwijana oznacza chwilowy brak dostępnych specjalistów lub specjalizacji.

Rysunek 21: Dodaj specjalistę

Wyświetl listę specjalistów - powoduje pobranie z bazy i wyświetlenie w formie listy wszystkich specjalistów. Z tego miejsca można wybrać opcje edycji danych specjalisty, aktywować lub dezaktywować go lub usunąć, jeżeli nie jest on powiązany z żadną rezerwacją. Możliwe jest także przejście do listy, która przedstawia wszystkie rezerwacje, jakie posiada dany specjalista. Lista widoczna jest na rysunku 22.

Lista specjalistów

• Operacja zakończyła się sukcesem.

Kod specjalizacji	Specjalizacja	Imię specjalisty	Nazwisko specjalisty	Numer PWZ	Akcje
001	Neurologia	Janina	Fornalska	1002001	<input type="button" value="Edytuj specjalistę"/> <input type="button" value="Aktywuj"/> <input type="button" value="Dezaktywuj"/> <input type="button" value="Usuń specjalistę"/> <input type="button" value="Rezerwacje"/>
001	Neurologia	Waldemar	Modrecki	1002002	<input type="button" value="Edytuj specjalistę"/> <input type="button" value="Aktywuj"/> <input type="button" value="Dezaktywuj"/> <input type="button" value="Usuń specjalistę"/> <input type="button" value="Rezerwacje"/>
002	Flebologia	Marian	Nowak	1231231	<input type="button" value="Edytuj specjalistę"/> <input type="button" value="Aktywuj"/> <input type="button" value="Dezaktywuj"/> <input type="button" value="Usuń specjalistę"/> <input type="button" value="Rezerwacje"/>

Rysunek 22: Wyświetl listę specjalistów

Aktywuj/Dezaktywuj specjalistę – dezaktywacja uniemożliwia wybór specjalisty przez pacjenta, aż do momentu jego ponownej aktywacji. Dezaktywacja jest możliwa nawet, jeżeli specjalista posiada powiązania z jakąkolwiek rezerwacją. Przypadek użycia zaprezentowany został na rysunku 22

Edytuj dane specjalisty – umożliwia edycje danych osobowych specjalisty poprzez formularz widoczny na rysunku 23.

Edycja specjalisty

Specjalizacja

Neurologia

Imię specjalisty

Janina

Nazwisko specjalisty

Fornalska

Numer PWZ

1002001

Wyślij

Anuluj

Rysunek 23: Edytuj dane specjalisty

Usuń specjalistę - umożliwia pracownikowi recepcji usunięcie specjalisty z bazy, jeżeli tylko nie został on powiązany z żadną rezerwacją. Usunięcie odbywa się poprzez wciśnięcie przycisku „Usuń specjalistę” widocznego na rysunku 22.

Wyświetl listę rezerwacji specjalisty - funkcja dostępna za pomocą przycisku „Rezerwacje” dostępnego na liście specjalistów na rysunku 22. Powoduje wyświetlenie wszystkich rezerwacji do których przydzielono danego specjalistę, widoczną na rysunku 24. Lista umożliwia usuwanie aktualnych i przeszłych rezerwacji, również masowo poprzez przyciski dostępne na dole strony.

Lista rezerwacji specjalisty

Specjalizacja	Specjalista	Data rezerwacji	Wybrana godzina	Rezerwujący	Numer tel. rezerwującego	Akcje
Neurologia	Janina Fornalska	15 listopada 2019	10:00	patient1	0426850421	Usuń rezerwację
Neurologia	Janina Fornalska	16 listopada 2019	08:00	patient1	0426850421	Usuń rezerwację
Neurologia	Janina Fornalska	11 grudnia 2019	08:00	patient	801696969	Usuń rezerwację
Neurologia	Janina Fornalska	2 lutego 2020	10:00	patient1	0426850421	Usuń rezerwację

Usuń wszystkie rezerwacje Powrót do strony głównej

Rysunek 24: Wyświetl listę rezerwacji specjalisty

Wyświetl listę wszystkich aktualnych rezerwacji - ten przypadek użycia jest dostępny dla pracownika recepcji. Polega on na pobraniu z bazy i wyświetleniu wszystkich aktualnych rezerwacji stworzonych przez pacjentów. Możliwe jest odwołanie pojedynczej rezerwacji. Listę zaprezentowano na rysunku 25.

Lista aktualnych rezerwacji

Specjalizacja	Specjalista	Data rezerwacji	Wybrana godzina	Rezerwujący	Numer tel. rezerwującego	Akcje
Flebologia	Marian Nowak	24 grudnia 2019	08:00	patient2	426804567	Usuń rezerwację
Neurologia	Krystyna Pigulska	27 grudnia 2019	08:00	patient2	426804567	Usuń rezerwację

Powrót do strony głównej

Rysunek 25: Wyświetl listę wszystkich aktualnych rezerwacji.

Wyświetl listę wszystkich przeszłych rezerwacji – polega na pobraniu z bazy i wyświetleniu wszystkich przeszłych rezerwacji. Z tego miejsca można usunąć pojedynczą przeszłą rezerwację, jak również usunąć je wszystkie. Listę zaprezentowano na rysunku 26.

Lista przeszłych rezerwacji

Specjalizacja	Specjalista	Data rezerwacji	Wybrana godzina	Rezerwujący	Numer tel. rezerwującego	Akcje
Neurologia	Janina Formalska	15 listopada 2019	10:00	patient1	0426850421	Usuń rezerwację
Neurologia	Janina Formalska	16 listopada 2019	08:00	patient1	0426850421	Usuń rezerwację

[Usuń wszystkie przeszłe rezerwacje](#)
[Powrót do strony głównej](#)

Rysunek 26: Wyświetl listę wszystkich przeszłych rezerwacji

Utwórz rezerwację – funkcjonalność dla poziomu dostępu „Pacjent”. Aby dokonać rezerwacji wizyty należy najpierw wybrać specjalizację z listy, a następnie kliknąć przycisk „Wybierz” co pokazuje rysunek 27. Wtedy nastąpi przekierowanie na kolejną stronę tworzenia rezerwacji pokazaną na rysunku 28, na której w sposób analogiczny należy wybrać specjalistę. Ostatni krok, to wybór daty i godziny rezerwacji z wykorzystaniem interaktywnego kalendarza widocznego na rysunkach 29 i 30 i wybranie przycisku „Utwórz rezerwację”. Kliknięcie przycisku „Cofnij” przenosi pacjenta na poprzednią stronę tworzenia rezerwacji.

Nowa rezerwacja

Specjalizacja

Wybierz specjalizację:

[Wybierz](#) [Anuluj](#)

Pusta lista rozwijana oznacza chwilowy brak dostępnych specjalistów lub specjalizacji.

Rysunek 27: Utwórz rezerwację – wybór specjalizacji

Nowa rezerwacja

Specjalizacja

Neurologia

Specjalista

Wybierz specjalistę:

[Wybierz](#) [Cofnij](#)

Pusta lista rozwijana oznacza chwilowy brak dostępnych specjalistów lub specjalizacji.

Rysunek 28: Utwórz rezerwację – wybór specjalisty

rezerwacja

2019-12-20 08

[Utwórz rezerwację](#) [Cofnij](#)

Pusta lista rozwijana oznacza chwilowy brak dostępnych specjalistów lub specjalizacji.

Rysunek 29: Utwórz rezerwację – wybór daty

rezerwacja

12

2019-12-20 12

Utwórz rezerwację Cofnij

Pusta lista rozwijana oznacza chwilowy brak dostępnych specjalistów lub specjalizacji.

Rysunek 30: Utwórz rezerwację – wybór godziny

Edytuj zamówienie – opcja dostępna dla pacjenta kliniki. Umożliwia ona zmianę daty wybranej rezerwacji, przypadek prezentuje rysunek 31.

Edytuj rezerwację

Specjalizacja

Neurologia

Specjalista

Janina Fornalska

Data rezerwacji [yyyy-MM-dd hh]

2019-12-02 08

Zapisz Anuluj

Wszystkie pola są wymagane.

Rysunek 31: Edytuj rezerwację

Wyświetl listę swoich aktualnych rezerwacji - ten przypadek użycia jest dostępny dla pacjenta i został pokazany na rysunku 32. Polega on na pobraniu z bazy i wyświetleniu wszystkich jego aktualnych rezerwacji. Możliwe jest usunięcie pojedynczej rezerwacji oraz jej edycja.

Lista aktualnych rezerwacji

Specjalizacja	Specjalista	Data rezerwacji	Wybrana godzina	Akcje	
Flebologia	Marian Nowak	17 grudnia 2019	08:00	Edytuj rezerwację	Usuń rezerwację
Neurologia	Janina Fornalska	20 grudnia 2019	12:00	Edytuj rezerwację	Usuń rezerwację
Neurologia	Krystyna Pigulska	2 stycznia 2020	08:00	Edytuj rezerwację	Usuń rezerwację

Powrót do strony głównej

Rysunek 32: Wyświetl listę swoich aktualnych rezerwacji

Wyświetl listę swoich przeszłych rezerwacji - pobiera z bazy i wyświetla wszystkie przeszłe rezerwacje danego pacjenta. Listę przedstawiono na rysunku 33. Pacjent nie ma możliwości usuwania, ani edytowania swoich przeszłych rezerwacji.

Lista przeszłych rezerwacji

Specjalizacja	Specjalista	Data rezerwacji	Wybrana godzina
Neurologia	Janina Fornalska	15 listopada 2019	10:00
Neurologia	Janina Fornalska	16 listopada 2019	08:00

[Powrót do strony głównej](#)

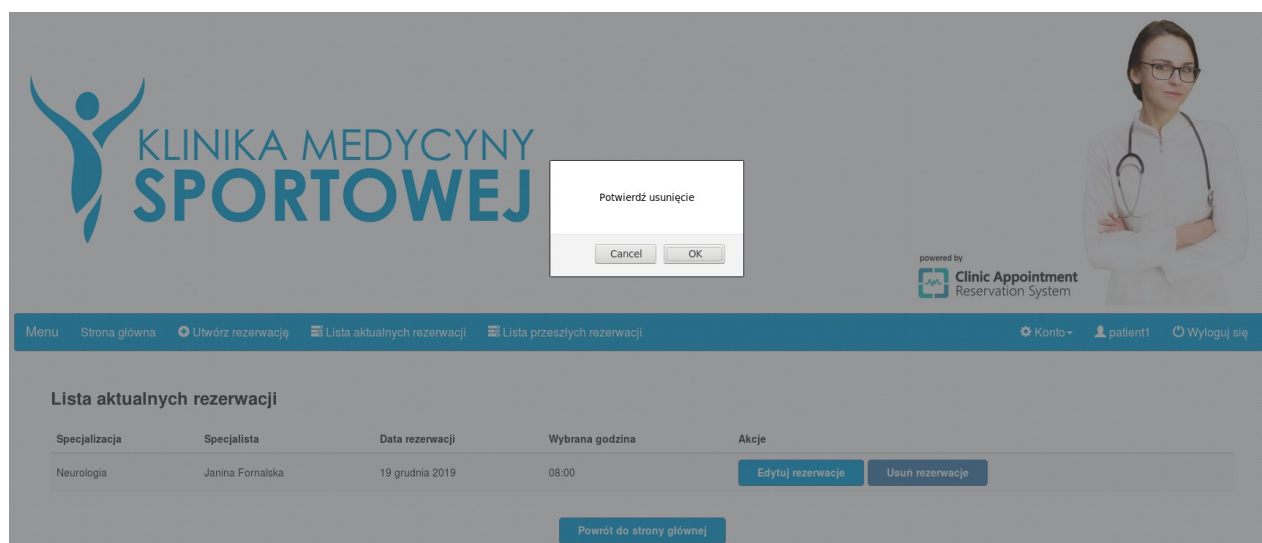
Rysunek 33: Wyświetl listę swoich przeszłych rezerwacji

Usuń swoją rezerwację – opcja dostępna dla pacjenta. Umożliwia usunięcie wybranej aktualnej rezerwacji z listy. Przycisk tej akcji pokazany został na rysunku 32. Akcja usuwania wymaga potwierdzenia w wyskakującym oknie pokazanym na rysunku 34.

Usuń dowolną rezerwację – umożliwia pracownikowi recepcji usunięcie dowolnej rezerwacji danego pacjenta. Przyciski tej akcji pokazane zostały na rysunkach 25 i 26. Akcja usuwania wymaga potwierdzenia w wyskakującym oknie pokazanym na rysunku 34.

Usuń wszystkie rezerwacje – funkcja dostępna jedynie dla pracowników recepcji. Po kliknięciu na przycisk 24 pokazanego na rysunku „Usuń wszystkie rezerwacje” oraz po potwierdzeniu tej operacji, zostają one bezpowrotnie usunięte. Akcja usuwania wymaga potwierdzenia w wyskakującym oknie pokazanym na rysunku 34.

Usuń wszystkie przeszłe rezerwacje – daje pracownik recepcji możliwość usunięcia wszystkich przeszłych rezerwacji za pomocą przycisku widocznego na rysunku 26. Akcja usuwania wymaga potwierdzenia w wyskakującym oknie pokazanym na rysunku 34.



Rysunek 34: Wyskakujące okno do potwierdzania akcji usuwania zamówień

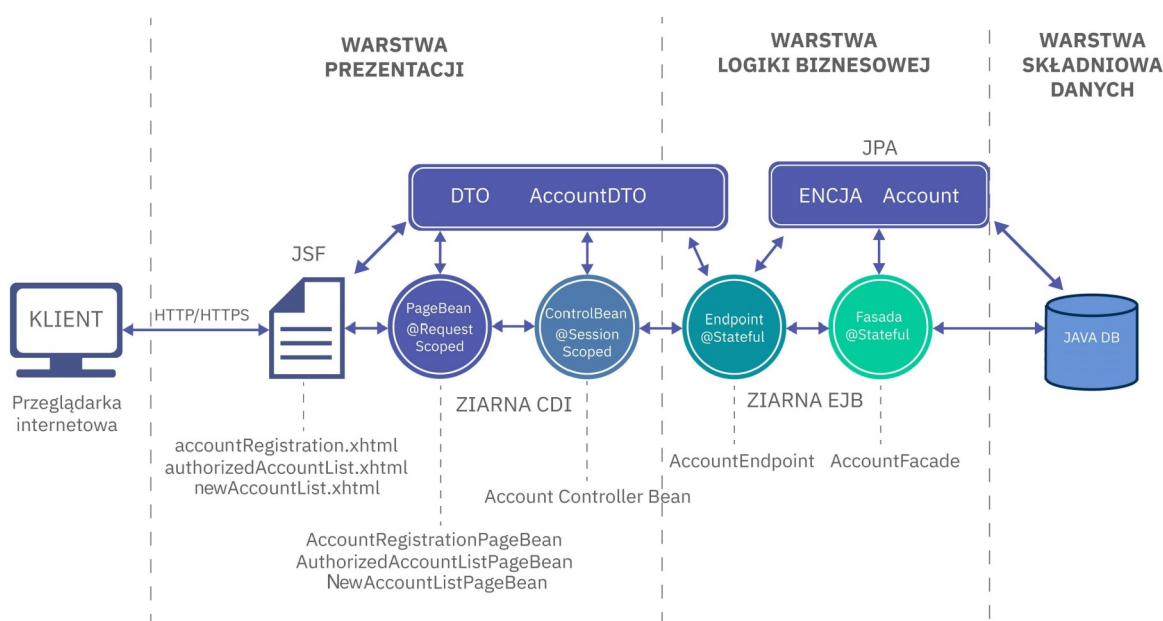
3 Realizacja projektu

W niniejszym rozdziale zostały zaprezentowane aspekty techniczne dotyczące projektowania i tworzenia systemu. Przedstawione zostały struktury poszczególnych warstw aplikacji, funkcjonalność systemu, obsługa wyjątków aplikacyjnych, bezpieczeństwo, kontrola odpowiedzialności oraz instrukcja wdrożenia systemu.

3.1 Realizacja przykładowego CRUD'a

CRUD (ang. create, read, update, delete) to akronim odzwierciedlający podstawowe funkcje umożliwiające zarządzanie danymi w bazie danych [8]. Są one stosowane do tworzenia lub dodania nowych informacji (create), odczytania lub wyświetlenia istniejących informacji (read), modyfikowania lub edycji istniejących informacji (update) lub też usuwania istniejących informacji (delete).

Poniżej znajduje się szczegółowy CRUD, dotyczący różnych aspektów związanych z obsługą kont użytkowników oraz odpowiadający mu diagram komponentów - Rysunek 35.



Rysunek 35: Diagram komponentów dla przypadków użycia CRUD związanych z kontami użytkowników.

3.1.1 Tworzenie

CREATE – tworzenie (zarejestrowanie) konta. Celem tego przypadku jest utworzenie nowego konta użytkownika poprzez dodanie nowej pozycji w tabeli ACCOUNT w bazie danych. W tym celu nieuwierzytelniony użytkownik aplikacji wypełnia formularz widoczny na rysunku 5, znajdujący się na stronie *accountRegistration.xhtml*, której fragment kodu zaprezentowano na listingu 1.

```

<label class="text-left">
    <h:outputLabel value="#{msg['page.register.form.label.name']}" " />
</label>
<h:inputText class="margin-bottom-small" id="name" maxlength="60"
    required="true" validatorMessage="#{msg['page.account.validator.name']}"
    value="${accountRegistrationPageBean.accountDTO.name}" >
    <f:validateLength minimum="2" maximum="60" ></f:validateLength>
    <f:validateRegex pattern="[A-ZĄĆĘŁŃÓŚŻ] [a-ząćęłńóśż]+([\s][A-ZĄĆĘŁŃÓŚŻ]
    [a-ząćęłńóśż]+)" />
</h:inputText>
<h:message for="name" styleClass="error_small"/>
(...)
<label class="text-left">
    <h:outputLabel value="#{msg['page.register.form.label.login']}" " />
</label>
<h:inputText class="margin-bottom-small" id="login" maxlength="20"
    required="true"
    validatorMessage="#{msg['page.account.validator.login']}"
    value="${accountRegistrationPageBean.accountDTO.login}" >
    <f:validateLength minimum="6" maximum="20" ></f:validateLength>
    <f:validateRegex pattern="[a-zA-Z0-9]{6,20}" />
</h:inputText>
<h:message for="login" styleClass="error_small"/>
<label class="text-left">
    <h:outputLabel value="#{msg['page.register.form.label.password']}" " />
</label>
<h:inputSecret class="margin-bottom-small" id="password" maxlength="30"
    required="true"
    validatorMessage="#{msg['page.account.validator.password']}"
    value="${accountRegistrationPageBean.accountDTO.password}" >
    <f:validateLength minimum="8" maximum="30" ></f:validateLength>
    <f:validateRegex pattern="(?=.*\d) (?=.*[a-ząćęłńóśż]) (?=.*[A-ZĄĆĘŁŃÓŚŻ])
    (?=.*[\W]).{8,30}" />
</h:inputSecret>
<h:message for="password" styleClass="error_small"/>

```

Listing 1: Fragment kodu strony registerAccount.xhtml - formularz odpowiedzialny za tworzenie konta użytkownika.

Po wypełnieniu wszystkich wymaganych pól formularza i wciśnięciu przycisku „Zarejestruj” zostaje wywołana metoda `registerAccountAction()` znajdująca się w klasie `AccountRegistrationPageBean` której fragment pokazuje listing 2. Metoda `registerAccountAction()` jest odpowiedzialna za sprawdzenie poprawności danych wprowadzonych w formularzu. W jej ciele wywoływana jest metoda `registerAccount()` klasy kontrolera `AccountControllerBean`, która jako parametr przyjmuje obiekt klasy `AccountDTO`.

```

@Named(value = "accountRegistrationPageBean")
@RequestScoped
public class AccountRegistrationPageBean implements Serializable {

    private AccountDTO accountDTO;
    (...)
    @PostConstruct
    public void init() {
        accountDTO = new AccountDTO();
    }

    public String registerAccountAction() {
        if (passwordRepeat.equals(accountDTO.getPassword())) {

```

```

        try {
            accountControllerBean.registerAccount(accountDTO);
        } catch (AppBaseException ex) {

Logger.getLogger(AccountRegistrationPageBean.class.getName()).log(Level.SEVERE
, null, ex);

            ContextUtils.emitI18NMessage(null, ex.getMessage());
            return null;
        }
    } else {
        ContextUtils.emitI18NMessage("RegisterForm:passwordRepeat",
"passwords.not.matching");
        return null;
    }
    return "main"; } }

```

Listing 2: Fragment kodu klasy AccountRegistrationPageBean.

Metoda `registerAccount()` klasy `AccountControllerBean` widocznej na listingu 3 ma za zadanie weryfikację, czy nie jest wywoływana dwukrotnie ta sama czynność. Sprawdza, czy ten sam obiekt DTO nie jest kolejny raz podawany jako parametr wykonania metody, aby te same dane nie zostały ponownie wprowadzane do bazy. W metodzie wykorzystano mechanizm, który sprawdza, czy nie doszło do odwołania realizowanej transakcji aplikacyjnej. Jeśli taka transakcja została odwołana, ma miejsce ponowne wykonanie metody. Gdy odwołanie nastąpi trzykrotnie, zostaje zgłoszony wyjątek

```

@Named(value = "accountControllerBean")
@SessionScoped
public class AccountControllerBean implements Serializable {

    @EJB
    private AccountEndpoint accountEndpoint; (...)
    public void registerAccount(final AccountDTO accountDTO) throws
AppBaseException {
        final int UNIQUE_METHOD_ID = accountDTO.hashCode() + 1;
        if (lastActionMethod != UNIQUE_METHOD_ID) {
            int endpointCallCounter =
accountEndpoint.NB_ATTEMPTS_FOR_METHOD_INVOCATION;
            do {
                accountEndpoint.registerAccount(accountDTO);
                endpointCallCounter--;
            } while (accountEndpoint.isLastTransactionRollback() &&
endpointCallCounter > 0);
            if (endpointCallCounter == 0) {
                throw
AppBaseException.createExceptionForRepeatedTransactionRollback();
            }
            ContextUtils.emitI18NMessage("success", "error.success");
        } else {
            ContextUtils.emitI18NMessage(null, "error.repeated.action");
        }
        ContextUtils.getContext().getFlash().setKeepMessages(true);
        lastActionMethod = UNIQUE_METHOD_ID;
    }
}

```

Listing 3: Fragment kodu klasy AccountControllerBean

Metoda `registerAccount(AccountDTO accountDTO)` klasy `AccountEndpoint` pokazanej na listingu 4, odpowiedzialna jest za przeniesienie danych z obiektu DTO na obiekt encji i wywołanie metody `create` w klasie `AccountFacade`.

```

@Stateful
@Transactional(TransactionalAttributeType.REQUIRES_NEW)
@Interceptors(LoggingInterceptor.class)
public class AccountEndpoint extends AbstractEndpoint implements
SessionSynchronization {

    @EJB
    private AccountFacade accountFacade;

    (...)

    @PermitAll
    public void registerAccount(AccountDTO accountDTO) throws AppBaseException {
        Account newAccount = new NewAccount();
        newAccount.setLogin(accountDTO.getLogin());
        newAccount.setPassword(accountDTO.getPassword());
        newAccount.setQuestion(accountDTO.getQuestion());
        newAccount.setAnswer(accountDTO.getAnswer());
        newAccount.setName(accountDTO.getName());
        newAccount.setSurname(accountDTO.getSurname());
        newAccount.setPhoneNumber(accountDTO.getPhoneNumber());
        newAccount.setActive(false);
        newAccount.setAuthorized(false);
        accountFacade.create(newAccount);
    }
}

```

Listing 4: Fragment kodu klasy AccountEndpoint odpowiedzialny za przeniesienie danych z obiektu DTO na obiekt Encji i wywołanie metody „create” w klasie AccountFacade.

Metoda create, której kod widzimy na 5 listingu, odpowiada za przeciążenie metody create i wywołanie jej z klasy abstrakcyjnej AbstractFacade, która jako typizowany parametr przyjmuje obiekt klasy encyjnej.

```

@Stateless
@Transactional(TransactionalAttributeType.MANDATORY)
public class AccountFacade extends AbstractFacade<Account> { (...)
    @PermitAll
    @Override
    public void create(Account entity) throws AppBaseException {
        try {
            super.create(entity);
        } catch (DatabaseException e) {
            if (e.getCause() instanceof SQLNonTransientConnectionException) {
                throw
AppBaseException.createExceptionDatabaseConnectionProblem(e);
            } else {
                throw AppBaseException.createExceptionDatabaseQueryProblem(e);
            }
        } catch (PersistenceException e) {
            final Throwable cause = e.getCause();
            if (cause instanceof DatabaseException &&
cause.getMessage().contains(DB_UNIQUE_CONSTRAINT_ACCOUNT_LOGIN)) {
                throw AccountException.createExceptionLoginAlreadyExists(e,
entity);
            } else {
                throw AppBaseException.createExceptionDatabaseQueryProblem(e);
            }
        }
    }
}

```

Listing 5: Fragment kodu klasy AccountFacade – metoda create.

W zaprezentowanej na listingu 6 klasie `AbstractFacade` wywoływana jest metoda `persist` na obiekcie klasy `EntityManager`, która powoduje utworzenie nowej krotki w tabeli `ACCOUNT` w bazie danych.

```
public abstract class AbstractFacade<T> {
    private Class<T> entityClass;

    public AbstractFacade(Class<T> entityClass) {
        this.entityClass = entityClass;
    }
    protected abstract EntityManager getEntityManager();

    public void create(T entity) throws AppBaseException {
        getEntityManager().persist(entity); //tu mamy Persista
        getEntityManager().flush();
    } (...)
```

Listing 6: Fragment kodu klasy `AbstractFacade`

3.1.2 Wyświetlanie

READ - odczytywanie danych z bazy. Funkcja ta powoduje pobranie z bazy danych zbioru kont o podanych właściwościach i wyświetlenie ich jako listy kont nieautoryzowanych użytkowników w formie tabeli pokazanej na rysunku 14 w widoku użytkownika z poziomem dostępu **Administrator**. Kod strony JSF jest widoczny na listingu 7.

```
<h:form id="AccountsForm" class="text-left reservation-form">
(...)
<h:dataTable width="100%" var="row" class="table table-striped" value="$
{listNewAccountsPageBean.dataModelAccounts}">
    <label class="text-left">
        <h:column id="name" class="padding" >
            <f:facet name="header"> ${msg['page.register.form.label.name']}
        </f:facet>
        <h:outputText value="#{row.name}" />
    </h:column>
</label>
<label class="text-left">
    <h:column id="surname">
        <f:facet name="header"> ${msg['page.register.form.label.surname']}
    </f:facet>
    <h:outputText value="#{row.surname}" />
</h:column>
</label>
<label class="text-left">
    <h:column id="login">
        <f:facet name="header"> ${msg['page.register.form.label.login']}
    </f:facet>
    <h:outputText value="#{row.login}" />
</h:column>
</label>
```

Listing 7: Fragment kodu strony `listNewAccounts.xhtml` odpowiedzialnego za wyświetlenie listy wszystkich nieautoryzowanych kont.

Klasa `ListNewAccountsPageBean` pokazana na listingu 8 jest odpowiedzialna za wywołanie metody inicjującej `listNewAccounts()`, która przetwarza dane pobrane z klasy punktu dostępowego i przekazuje je stronie *JSF* w formie klasy `DataModel`..

```
@Named(value = "listNewAccountsPageBean")
@ViewScoped
public class ListNewAccountsPageBean implements Serializable {
    @EJB
    private AccountEndpoint accountEndpoint;
    @Inject
    private AccountControllerBean accountControllerBean;
    private List<AccountDTO> listAccounts;
    private List<AccessLevel> listAccessLevels;
    public List<AccessLevel> getListAccessLevels() {
        return listAccessLevels;
    }
    private DataModel<AccountDTO> dataModelAccounts;

    public ListNewAccountsPageBean() {
    }
    public DataModel<AccountDTO> getDataModelAccounts() {
        return dataModelAccounts;
    }
    @PostConstruct
    public void initListNewAccounts() {
        try {
            listAccounts = accountControllerBean.listNewAccounts();
        } catch (AppBaseException ex) {

            Logger.getLogger(ListNewAccountsPageBean.class.getName()).log(Level.SEVERE,
            null, ex);

            ContextUtils.emitI18NMessage(null, ex.getMessage());
        }
        dataModelAccounts = new ListDataModel<>(listAccounts);

        AccessLevel[] listAllAccessLevels = AccessLevel.values();
        for (AccessLevel accessLevel : listAllAccessLevels) {

            accessLevel.setAccessLevelI18NValue(ContextUtils.getI18NMessage(accessLevel.ge
            tAccessLevelKey()));
        }
        listAccessLevels = new
        ArrayList<>(Arrays.asList(listAllAccessLevels));
        listAccessLevels.remove(AccessLevel.ACCOUNT);
        listAccessLevels.remove(AccessLevel.NEWACCOUNT);
    }
}
```

Listing 8: Fragment kodu klasy `ListNewAccountsPageBean` odpowiedzialny za przetworzenie i wyświetlenie listy wszystkich nieautoryzowanych kont.

Po wywołaniu metody kontrolera `listNewAccounts()`, widocznej na listingu, następuje wywołanie metody `listNewAccounts()` klasy `AccountEndpoint`, której wynik zostaje przypisany i zapamiętany w zmiennej stanowej `selectedAccountsListsDTO` klasy `AccountDTO` i zwrócony w formie listy klasie `ListNewAccountPageBean`.

```
public List<AccountDTO> listNewAccounts() throws AppBaseException {
    int endpointCallCounter =
    accountEndpoint.NB_ATEMPTS_FOR_METHOD_INVOCATION;
    do {
```

```

        selectedAccountsListsDTO = accountEndpoint.listNewAccounts();
        endpointCallCounter--;
    } while (accountEndpoint.isLastTransactionRollback() &&
endpointCallCounter > 0);
    if (endpointCallCounter == 0) {
        throw
AppBaseException.createExceptionForRepeatedTransactionRollback();
    }
    return selectedAccountsListsDTO;
}

```

Listing 9: Fragment kodu klasy AccountControllerBean – metoda listNewAccounts().

W dalszej kolejności dane pobierane są dzięki metodzie listNewAccounts() klasy AccountEndpoint, widocznej na listingu 10, odpowiada ona za wywołanie metody findNewAccount() klasy AccountFacade i przepisanie zwróconych przez nią danych w formacie obiektu listy encyjnej na obiekt typu listy DTO.

```

@RolesAllowed({"Admin"})
public List<AccountDTO> listNewAccounts() throws AppBaseException {
    List<Account> listRegisteredAccount = accountFacade.findNewAccount();
    savedAccountStateList = listRegisteredAccount;
    List<AccountDTO> listNewRegisteredAccount = new ArrayList<>();
    for (Account account : listRegisteredAccount) {
        AccountDTO accountDTO = new AccountDTO(
            account.getLogin(),
            account.getName(),
            account.getSurname(),
            account.getPhoneNumber(),
            account.getCreatedAt()
        );
        listNewRegisteredAccount.add(accountDTO);
    }
    Collections.sort(listNewRegisteredAccount);
    return listNewRegisteredAccount;
}

```

Listing 10: Fragment kodu klasy AccountEndpoint

Metoda findNewAccount() której kod został zaprezentowany na listingu 11 na celu wywołanie kwerendy wyszukującej Account.findNewAccount i przekazanie zwróconych przez nią danych w formie listy do klasy AccountEndpoint.

```

@RolesAllowed({"Admin"})
public List<Account> findNewAccount() throws AppBaseException {
    TypedQuery<Account> tq = em.createNamedQuery("Account.findNewAccount",
Account.class);
    try {
        return tq.getResultList();
    } catch (PersistenceException e) {
        final Throwable cause = e.getCause();
        if (cause instanceof DatabaseException && cause.getCause()
instanceof SQLNonTransientConnectionException) {
            throw
AppBaseException.createExceptionDatabaseConnectionProblem(e);
        } else {
            throw
AppBaseException.createExceptionDatabaseQueryProblem(cause);
        } } }

```

Listing 11: Metoda klasy AccountFacade

Na listingu 12 widnieje kwerenda `Account.findNewAccount`, która jest odpowiedzialna za pobranie z bazy danych z tabeli `ACCOUNT` wszystkich nieautoryzowanych kont użytkowników. Odbywa się to dzięki zastosowaniu w projekcie strategii `SINGLE_TABLE`, która umożliwia podzielenie wszystkich obiektów kont na osobne *subklasy* - w tym wypadku pobieramy wszystkie konta klasy `NewAccount`.

```
@NamedQuery(name = "Account.findNewAccount", query = "SELECT a FROM
    NewAccount a")
```

Listing 12: Kwerenda klasy encyjnej Account.

3.1.3 Edycja

UPDATE - aktualizowanie danych w bazie. Za pośrednictwem tej funkcji ustawiany jest status aktywności konta na aktywne (wartość `true`) lub nieaktywne (wartość `false`). Jedynie na aktywne konto można się uwierzytelnić. Przycisk do aktywacji znajduje się na zaprezentowanej na rysunku 15 stronie `listAuthorizedAccounts.xhtml`, dostępnej z poziomu dostępu **Administrator**. Kod strony JSF został zaprezentowany na listingu 13.

```
<h:commandButton value="${msg['page.authorized.accounts.list.active']}"
    class="main-button margin-right-small"
    action="#{listAuthorizedAccountsPageBean.activateAccountAction(row)}"
    disabled="#{row.active}" />
<h:commandButton value="${msg['page.authorized.accounts.list.deactive']}"
    class="main-button"
    action="#{listAuthorizedAccountsPageBean.deactivateAccountAction(row)}"
    disabled="#{not row.active}" />
```

Listing 13: Fragment kodu strony `listAuthorizedAccount.xhtml` odpowiedzialny za wyświetlenie przycisków umożliwiających aktywację i dezaktywację konta autoryzowanego użytkownika.

W celu aktywacji konta, należy wybrać z tabeli odpowiedni przycisk „Aktywuj konto”, odpowiedzialny za wywołanie metody `activateAccountAction(row)` znajdującej się w przedstawionej na listingu 14 klasie `ListAuthorizedAccountsPageBean`. Wywołana metoda przekazuje do kolejnej metody kontrolera klasy `AccountControllerBean` wskazany jako parametr obiekt klasy `AccountDTO`. Po aktualizacji danych w bazie następuje odświeżenie widoku tabeli.

```
@Named(value = "listAuthorizedAccountsPageBean")
@ViewScoped
public class ListAuthorizedAccountsPageBean implements Serializable {
    (...)
    public String activateAccountAction(AccountDTO accountDTO) {
        try {
            accountControllerBean.activateAccount(accountDTO);
        } catch (AppBaseException ex) {
            Logger.getLogger(ListAuthorizedAccountsPageBean.class.getName())
                .log(Level.SEVERE, null, ex);
            ContextUtils.emitI18NMessage(null, ex.getMessage());
        }
        initListAuthorizedAccounts();
    }
}
```

```

        return null;
    public String deactivateAccountAction(AccountDTO accountDTO) {
        try {
            accountControllerBean.deactivateAccount(accountDTO);
        } catch (AppBaseException ex) {
            Logger.getLogger(ListAuthorizedAccountsPageBean.class.getName())
                .log(Level.SEVERE, null, ex);
            ContextUtils.emitI18NMessage(null, ex.getMessage());
        }
        initListAuthorizedAccounts();
        return null;
    }
}

```

Listing 14: Fragment kodu klasy `ListAuthorizedAccountPageBean` odpowiedzialny za wywołanie metod aktywacji i dezaktywacji konta.

Metoda `activateAccount()` klasy `AccountControllerBean`, widocznej na listingu 15 ma za zadanie zweryfikować, czy nie jest wywoływana dwukrotnie ta sama czynność. Sprawdza, czy ten sam obiekt DTO nie jest kolejny raz podawany jako parametr wykonania metody, aby te same dane nie były ponownie aktualizowane w bazie. W przypadku pozytywnej weryfikacji następuje wywołanie metody `activateAccount()` z parametrem `accountDTO.getLogin()` znajdującej się w klasie `AccountEndpoint`.

```

public void activateAccount(final AccountDTO accountDTO) throws
AppBaseException {
    final int UNIQ_METHOD_ID = accountDTO.hashCode() + 5;
    if (lastActionMethod != UNIQ_METHOD_ID) {
        int endpointCallCounter =
accountEndpoint.NB_ATEMPTS_FOR_METHOD_INVOCATION;
        do {
            accountEndpoint.activateAccount(accountDTO.getLogin());
            endpointCallCounter--;
        } while (accountEndpoint.isLastTransactionRollback() &&
endpointCallCounter > 0);
        if (endpointCallCounter == 0) {
            throw
AppBaseException.createExceptionForRepeatedTransactionRollback();
        }
        ContextUtils.emitI18NMessage("AccountsForm:success",
"error.success");
    } else {
        ContextUtils.emitI18NMessage(null, "error.repeated.action");
    }
    ContextUtils.getContext().getFlash().setKeepMessages(true);
    lastActionMethod = UNIQ_METHOD_ID;
}

```

Listing 15: Metoda kontrolera `AccountControllerBean` odpowiedzialna za sprawdzenie czy metoda aktywacji konta nie została wykonana ponownie.

W klasie `AccountEndpoint` widocznej na listingu 16 następuje wywołanie metody `endpointa selectAccountWithIterator()`, z parametrami `login` i `savedAccountsStateList`, która z zapamiętanej wcześniej encyjnej listy stanowej zwraca po podanym parametrze obiekt encji z zapamiętanym stanem konta. Następnie do nowego obiektu encji poleceniem `setActive(true)` zostaje przypisana właściwość aktywacji konta, którą wprowadzamy do bazy danych poprzez aktualizację

znajdującego się w niej rekordu poleceniem `edit`. Jeśli konto nie było wcześniej aktywowane ani edytowane, to ta właściwość jest do niego przypisywana razem z informacją, który administrator wprowadził tą zmianę.

```
@RolesAllowed({"Admin"})
public void activateAccount(String login) throws AppBaseException {
    Account account = selectAccountWithIterator(login,
        savedAccountStateList);
    if (!adminFacade.findByLogin(sessionContext.getCallerPrincipal()
        .getName()).isActive()) {
        throw AccountException.createExceptionAccountNotActive(account);
    }
    if (!account.isActive()) {
        account.setActive(true);
        account.setModifiedBy(loadCurrentAdmin());
        accountFacade.edit(account);
    } else {
        throw
        AccountException.createExceptionAccountAlreadyActivated(account);
    }
}
```

Listing 16: Metoda klasy `AccountEndpoint` odpowiedzialna za wywołanie metody wyszukiującej `findByLogin` w klasie `AccountFacade` i ustawieniu stanu aktywności konta.

W klasie `AccountFacade` wywołanie metody `edit` powoduje sprawdzenie przekazywanych danych z danymi znajdującymi się w bazie danych i w razie niezgodności pola wersji lub niezalezieniu zgłoszenie wyjątków odpowiednio: `OptimisticLockException` lub `NoResultException` co pokazuje listing 17. Jeśli żadne problemy nie wystąpią, przekazane zmienione dane są utrwalane w tabeli bazy danych `ACCOUNT`.

```
@PermitAll
@Override
public void edit(Account entity) throws AppBaseException {
    try {
        super.edit(entity);
    } catch (DatabaseException e) {
        if (e.getCause() instanceof SQLNonTransientConnectionException) {
            throw
            AppBaseException.createExceptionDatabaseConnectionProblem(e);
        } else {
            throw AppBaseException.createExceptionDatabaseQueryProblem(e);
        }
    } catch (NoResultException e) {
        throw AccountException.createExceptionNoAccountFound(e);
    } catch (OptimisticLockException e) {
        throw AppBaseException.createExceptionOptimisticLock(e);
    } catch (PersistenceException e) {
        throw AppBaseException.createExceptionDatabaseQueryProblem(e);
    }
}
```

Listing 17: Metoda `edit` klasy `AccountFacade`.

3.1.4 Usuwanie

DELETE – usunięcie konta. Funkcja ta powoduje usunięcie z bazy danych wybranego z listy konta. Usunięcie dostępne jest w formie przycisku tylko na przedstawionej na rysunku 14 stronie `listNewAccounts.xhtml` ponieważ możliwość kasowania jest dopuszczalna tylko w przypadku nieautoryzowanych kont. Dostęp do tej funkcji ma administrator systemu.

W celu skasowania konta, należy wybrać w tabeli odpowiedni przycisk „Usuń konto”, który widoczny jest w kodzie na listingu 18, który jest odpowiedzialny za wywołanie metody `deleteSelectedAccountAction(row)` znajdującej się w klasie `ListNewAccountsPageBean`.

```
<h:commandButton value="{msg['page.new.accounts.list.action.delete']}" (...)  
action="{listNewAccountsPageBean.deleteSelectedAccountAction(row)}"/>  
}
```

Listing 18: Fragment kodu strony `listNewAccounts.xhtml` odpowiedzialny za wyświetlenie przycisku do kasowania konta.

Wywołana metoda pokazana na listingu 19 przekazuje do kolejnej metody kontrolera klasy `AccountControllerBean` wskazany jako parametr obiekt klasy `AccountDTO`. Po usunięciu danych z bazy następuje odświeżenie widoku tabeli.

```
public String deleteSelectedAccountAction(AccountDTO accountDTO) {  
    try {  
        accountControllerBean.deleteAccount(accountDTO);  
    } catch (AppBaseException ex)  
{  
        Logger.getLogger(NewAccountsListPageBean.class.getName()).log(Leve  
l.SEVERE, null, ex);  
        ContextUtils.emitI18NMessage(null, ex.getMessage());  
    }  
    initListNewAccounts();  
    return null;  
}
```

Listing 19: Fragment klasy `ListNewAccountsPageBean`.

Metoda `deleteAccount()` klasy `AccountControllerBean` przedstawionej na listingu 20 ma za zadanie sprawdzić, czy nie jest wywoływana dwukrotnie ta sama akcja, czyli, czy ten sam obiekt `DTO` nie jest ponownie podawany jako jej parametr wykonania. W tym wypadku – czy nie próbujemy dwukrotnie usunąć z bazy danych tego samego obiektu. W przypadku pozytywnej weryfikacji w klasie `AccountEndpoint` wywoływana jest metoda `deleteAccount()` z parametrem `accountDTO`.

```
public void deleteAccount(final AccountDTO accountDTO) throws  
AppBaseException {  
    final int UNIQ_METHOD_ID = accountDTO.hashCode() + 2;  
    if (lastActionMethod != UNIQ_METHOD_ID) {  
        int endpointCallCounter =  
accountEndpoint.NB_ATEMPTS_FOR_METHOD_INVOCATION;  
        do {  
            accountEndpoint.deleteAccount(accountDTO);  
            endpointCallCounter--;  
        }
```

```

        } while (accountEndpoint.isLastTransactionRollback() &&
endpointCallCounter > 0);
        if (endpointCallCounter == 0) {
            throw
AppBaseException.createExceptionForRepeatedTransactionRollback();
        }
        ContextUtils.emitI18NMessage("AccountsForm:success",
"error.success");
    } else {
        ContextUtils.emitI18NMessage(null, "error.repeated.action");
    }
    ContextUtils.getContext().getFlash().setKeepMessages(true);
    lastActionMethod = UNIQU_METHOD_ID;
}

```

Listing 20: Metoda klasy AccountControlerBean odpowiedzialna na sprawdzenie czy nie została podjęta próba ponownego wykasowania tego samego obiektu.

W znajdującej się na listingu 21 klasie AccountEndpoint następuje wywołanie metody klasy fasady AccountFacade findByLogin(), która zwraca obiekt encji konta. Następnie obiekt ten jest przekazany jako parametr w metodzie remove wywoływanej na tej samej fasadzie.

```

@RolesAllowed({"Admin"})
public void deleteAccount(AccountDTO accountDTO) throws AppBaseException {
    accountState = accountFacade.findByLogin(accountDTO.getLogin());
    if (accountState instanceof NewAccount) {
        accountFacade.remove(accountState);
    } else {
        throw
AccountException.ceateExceptionAccountChangedByAnotherAdmin(accountState);
    }
}

```

Listing 21: Metoda klasy AccoutEndpoint odpowiedzialna na wywołanie kwerendy wyszukującej obiekt z bazy danych oraz wywołująca polecenie remove mające na celu usunięcie tego obiektu.

W klasie AccountFacade następuje przeciążenie pokazanej na listingu 22 metody remove i próba wykrycia przyczyny potencjalnego wyjątku bazodanowego. Jeśli żadne wyjątki nie zostaną zgłoszone, przy pomocy polecenia super następuje wywoływana metody w nadrzędnej klasie abstrakcyjnej AbstractFacade. Metoda ta, jako typizowany parametr, przyjmuje obiekt klasy encyjnej.

```

@RolesAllowed({"Admin"})
@Override
public void remove(Account entity) throws AppBaseException {
    try {
        super.remove(entity);
    } catch (DatabaseException e) {
        if (e.getCause() instanceof SQLNonTransientConnectionException) {
            throw
AppBaseException.createExceptionDatabaseConnectionProblem(e);
        } else {
            throw AppBaseException.createExceptionDatabaseQueryProblem(e);
        }
    } catch (OptimisticLockException e) {
        throw AppBaseException.createExceptionOptimisticLock(e);
    }
}

```



```

        } catch (PersistenceException e) {
            final Throwable cause = e.getCause();
            if (cause instanceof DatabaseException &&
                (cause.getMessage().contains(DB_FK_SPECIALIZATION_SPECIALIZATION_CREATOR) ||
                 cause.getMessage().contains(DB_FK_SPECIALIZATION_MODIFIED_BY))) {
                throw
AccountException.createExceptionAccountInUseInSpecialization(e, entity);
            }
            if (cause instanceof DatabaseException &&
                (cause.getMessage().contains(DB_FK_SPECIALIST_SPECIALIST_CREATOR) ||
                 cause.getMessage().contains(DB_FK_SPECIALIST_MODIFIED_BY))) {
                throw
AccountException.createExceptionAccountInUseInSpecialist(e, entity);
            }
            if (cause instanceof DatabaseException &&
                (cause.getMessage().contains(DB_FK_ACCOUNT_ACCOUNT_CREATOR) ||
                 cause.getMessage().contains(DB_FK_ACCOUNT_MODIFIED_BY))) {
                throw AccountException.createExceptionAccountInUseInAccount(e,
entity);
            }
            if (cause instanceof DatabaseException &&
                (cause.getMessage().contains(DB_FK_RESERVATION_RESERVATION_CREATOR) ||
                 cause.getMessage().contains(DB_FK_RESERVATION_MODIFIED_BY))) {
                throw
AccountException.createExceptionAccountInUseInReservation(e, entity);
            } else {
                throw AppBaseException.createExceptionDatabaseQueryProblem(e);
            }
        }
    }
}

```

Listing 22: Metoda klasy AccountFacade odpowiedzialna za obsługę wyjątków i usunięcie odpowiedniej krotki z bazy danych

W klasie zaprezentowanej na listingu 23 AbstractFacade, na obiekcie klasy EntityManager, wywoływana jest metoda remove, która powoduje usunięcie z tabeli ACCOUNT w bazie danych wybranego konta.

```

public void remove(T entity) throws AppBaseException {
    getEntityManager().remove(getEntityManager().merge(entity));
    getEntityManager().flush();
}

```

Listing 23: Metoda klasy AccountFacade odpowiedzialna za usunięcie z bazy danych wybranej krotki

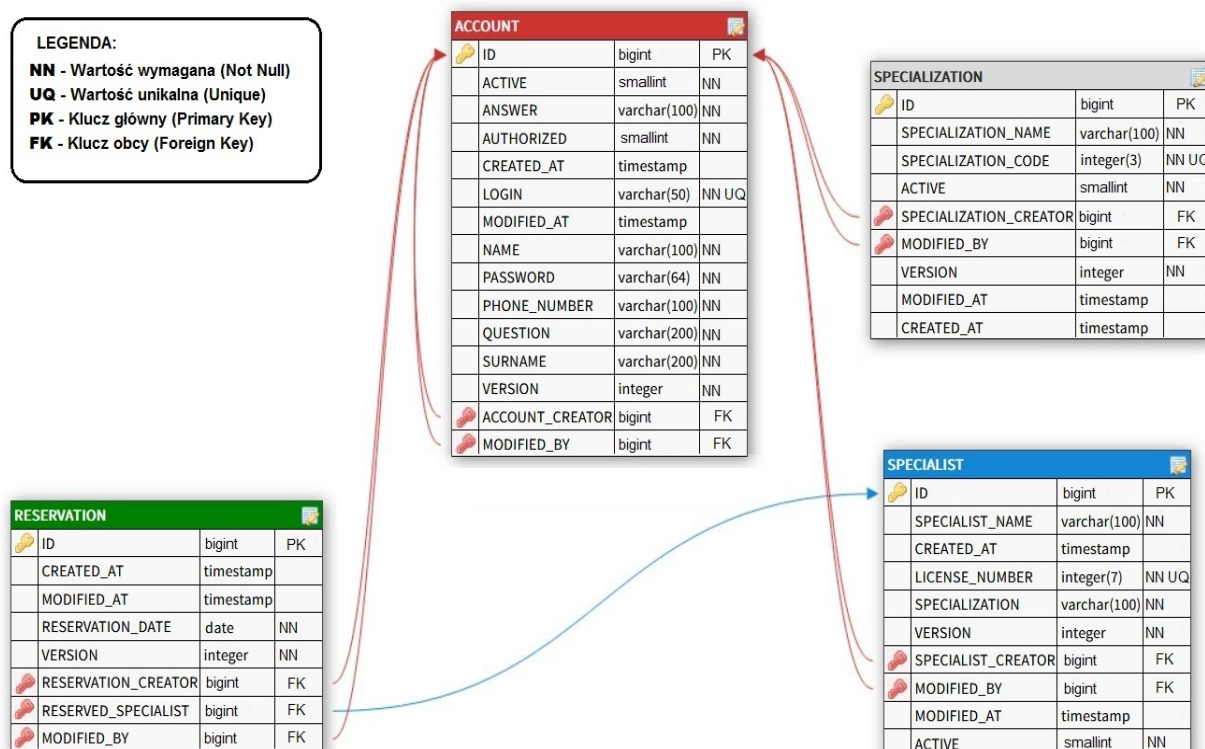
3.2 Warstwa składowania danych

Aplikacja korzysta z relacyjnej bazy danych zarządzanej przez JavaDB. To w niej składowane są dane aplikacji. Struktury bazy zbudowane są z tabel z danymi powiązanymi ze sobą za pomocą kluczy głównych i obcych unikalnych pól poszczególnych tabel [3]. Wartość klucza głównego nowego rekordu generowana jest przy pomocy generatora tabel @TableGenerator przedstawionego w listingu 26. Wszystkie dane, za których

wprowadzanie do bazy danych jest odpowiedzialny użytkownik aplikacji są wymagane [3] (ang. *not null*).

3.2.1 Model relacyjnej bazy danych

Rysunek 36 prezentuje diagram klas encyjnych, który prezentuje budowę i powiązania występujące pomiędzy obiektami w przedstawionym modelu danych. Model ten obejmuje encje wykonane w standardzie JPA (Java Persistence API). Ponadto wykorzystano również język zapytań JPQ [2], który służy do wyrażania zapytań operacyjnych na obiektach (encjach). Warunek wyszukiwania np. `SELECT a FROM Account a WHERE a.authorized = true`, zaprezentowany w listingu 26 umożliwia zawężenie zbioru wyników do kont, które posiadają parametr aktywności o wartości prawda (ang. *true*). Istotnym jest, że w języku JPQ odwołujemy się tylko do elementu encji, nie zaś do tabel bazy danych.



Rysunek 36: Relacyjny model bazy danych

3.2.2 Konfiguracja zasobów w relacyjnej bazie danych

Za konfigurację zasobów w relacyjnej bazie danych odpowiedzialny jest deskryptor składowania `presistance.xml`, widoczny na listingu 24. Definiuje on jednostkę składowania korzystającą z zasobu JDBC [9]. Do nawiązania połączenia z bazą danych aplikacja korzysta z puli połączeń JDBC [2] (ang. *JDBC Connection Pool*) i powiązanego z nią zasobu JDBC [8] (ang. *JDBC Resource*) pokazanych na listingu 24.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="CAR_PU" transaction-type="JTA">
    <jta-data-source>jdbc/CAR_DS</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <validation-mode>NONE</validation-mode>
    <properties>
      <property name="javax.persistence.schema-generation.database.action"
value="create"/>
      <property name="eclipselink.logging.level" value="FINE"/>
    </properties>
  </persistence-unit>
</persistence>
```

Listing 24: Plik deskryptora składowania persistence.xml

3.2.3 Mapowanie obiektowo-relacyjne ORM

Aplikacja działa na modelu obiektowym, który opiera się na klasach Javy i powiązanych z nimi relacyjnych tabelach bazy danych. Bazy te opierają swój mechanizm działania na relacjach. Oznacza to, że relacja jest zbiorem rekordów, w którym każdy z nich, może zawierać atrybuty określonych typów. Stosując jednak bardziej popularne nazewnictwo można powiedzieć, że relacje w relacyjnych bazach danych to tabele, które zawierają w sobie dane. Założeniem mechanizmu ORM [14] (ang. *Object Relational Mapping*) jest więc powiązanie ze sobą określonych obiektów, które zawierają proste dane, jak również odwołania do innych obiektów, z rekordami, które występują w obrębie tabel (relacji) [14].

Do utworzenia mapowania między obiema strukturami została wykorzystana specyfikacja Java Persistence API [2], która odpowiedzialna jest za połączenia z bazą danych, zapytania SQL [3] (ang. *Structured Query Language*) w postaci kwerend klas enyjnych, oraz mapowanie wyników tych zapytań na określone pola w aplikacji, z zastosowaniem odpowiednich adnotacji `@Column` widocznych na listingu 25 oraz 26.

Klasa `AbstractEntity` przedstawiona na listingu 25, to nadrzędna klasa encyjna, która implementuje część mechanizmu blokad optymistycznych, korzystając z deklaracji pola `Version` oraz mapuje wspólne dla wszystkich klas pola `createdAt` i `modifiedAt` na odpowiadające im kolumny `CREATED_AT` i `MODIFIED_AT` w relacyjnej bazie danych. Pola te informują o dacie utworzenia i ostatniej modyfikacji wszelkich wpisów w bazie danych.

```
@MappedSuperclass
public abstract class AbstractEntity {

    private static final long serialVersionUID = 1L;

    @Basic(optional = false)
    @NotNull
    @Column(name = "VERSION", nullable = false)
    @Version
    private int version;

    @Basic(optional = false)
```

```

@Column(name = "CREATED_AT", nullable = true, updatable = false)
@Temporal(TemporalType.TIMESTAMP)
private Date createdAt;

@Basic(optional = false)
@Column(name = "MODIFIED_AT", nullable = true)
@Temporal(TemporalType.TIMESTAMP)
private Date modifiedAt;
(...)
@PrePersist
private void prePersist() {
    createdAt = new Date();
}
@PreUpdate
private void preUpdate() {
    modifiedAt = new Date();
}
(...)
```

Listing 25: Klasa AbstractEntity - Implementacja bazowej klasy dla encji

Na listingu 30 zaprezentowano część klasy encyjnej Account. Klasa ta, wykorzystując strategię dziedziczenia pojedynczej tabeli [2] (ang. Single table) poprzez kolumnę dyskryminatora o nazwie LEVEL_OF_ACCESS, która posiada adnotację @DiscriminatorColumn, łączy klasy Admin, Reception, Patient oraz NewAccount. W efekcie, pięć klas encyjnych zostaje rozszerzonych przez klasę Account, a ich obiekty odwzorowywane są na rekordy w tabeli nadrzędnej ACCOUNT w bazie danych.

```

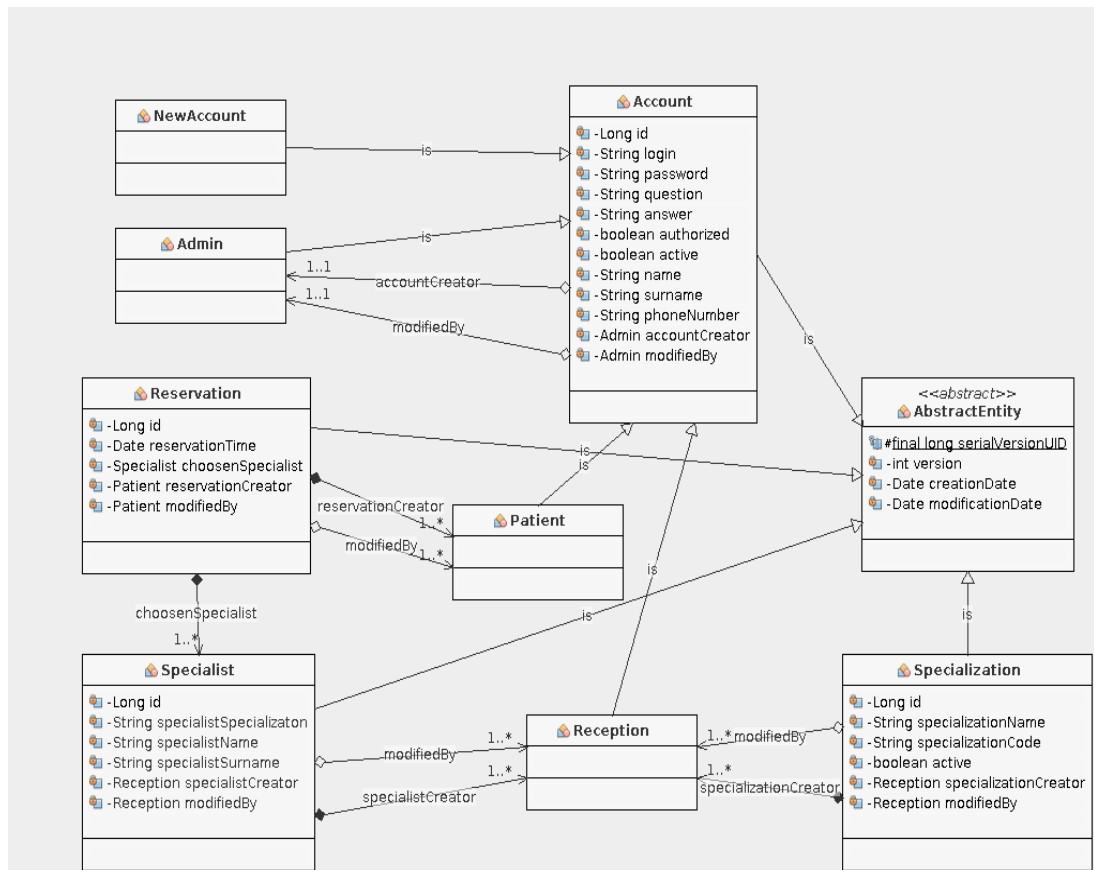
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "LEVEL_OF_ACCESS", discriminatorType =
DiscriminatorType.STRING)
@Table(name = "ACCOUNT", uniqueConstraints = {
    @UniqueConstraint(name = "UNIQUE_LOGIN", columnNames = "LOGIN")
})
@TableGenerator(name = "AccountGenerator", table = "TableGenerator",
    pkColumnName = "ID", valueColumnName = "value", pkColumnValue =
    "AccountGen")
@NamedQueries({
    @NamedQuery(name = "Account.findAll", query = "SELECT a FROM Account a"),
    @NamedQuery(name = "Account.findByLogin", query = "SELECT a FROM Account a
        WHERE a.login = :login"),
    @NamedQuery(name = "Account.findNewAccount", query = "SELECT a FROM
        NewAccount a"),
    @NamedQuery(name = "Account.findPatientAccounts", query = "SELECT a FROM
        Patient a"),
    @NamedQuery(name = "Account.findAuthorizedAccount", query = "SELECT a FROM
        Account a WHERE a.authorized = true")
})
public class Account extends AbstractEntity implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator =
        "AccountGenerator")
    @Basic(optional = false)
    @NotNull
    @Column(name = "ID", updatable = false)
```

```
protected Long id;
(...)
```

Listing 26: Fragment klasy encyjnej Account.

Na rysunku 37 przedstawiono diagram klas encyjnych, który przedstawia strukturę i powiązania, jakie zachodzą między obiektami w prezentowanym modelu danych. Model ten uwzględnia encje wykonane w standardzie JPA (Java Persistence API).



Rysunek 37: Diagram klas encyjnych.

3.3 Warstwa logiki biznesowej

Do zaimplementowania warstwy logiki biznesowej wykorzystano technologię Java EE, komponenty i kontener EJB [4] (ang. *Enterprise JavaBeans*). Warstwa ta jest podzielona na dwie podwarstwy:

- warstwę manipulacji danymi – posiada bezstanowe komponenty EJB, które implementują wzorzec fasady dla encji.
- warstwę właściwą logiki – zawiera stanowe komponenty EJB, które zapewniają transakcyjność;

3.3.1 Komponenty EJB

Sesyjne, bezstanowe komponenty EJB oznaczone adnotacją `@Stateless`, reprezentują fasady [4] (ang. *facade*). Zawierają one metody, które realizują operacje na relacyjnej bazie danych. Klasą nadrzędną dla wszystkich fasad jest zaprezentowana na listingu 27, klasa `AbstractFacade`, która oferuje podstawowe metody do komunikacji z bazą danych. Wszystkie klasy jej potomne, nadpisują jej metody oraz uzupełniają je o możliwe występujące w danym przypadku użycia wyjątki, co widoczne jest na przykładzie fragmentu klasy `SpecialistFacade` na listingu 28.

```
public abstract class AbstractFacade<T> {
    private Class<T> entityClass;

    public AbstractFacade(Class<T> entityClass) {
        this.entityClass = entityClass;
    }
    protected abstract EntityManager getEntityManager();
    public void create(T entity) throws AppBaseException {
        getEntityManager().persist(entity);
        getEntityManager().flush();
    }
    public void edit(T entity) throws AppBaseException {
        getEntityManager().merge(entity);
        getEntityManager().flush();
    }
    public void remove(T entity) throws AppBaseException {
        getEntityManager().remove(getEntityManager().merge(entity));
        getEntityManager().flush();
    }
}
```

Listing 27: Fragment klasy abstrakcyjnej AbstractFacade.

```
@Stateless
@Transactional(TransactionalAttributeType.MANDATORY)
public class SpecializationFacade extends AbstractFacade<Specialization> {

    static final public String DB_UNIQUE_CONSTRAINT_SPECIALIZATION_CODE =
"UNIQUE_SPECIALIZA";
    static final public String DB_FK_SPECIALIST_SPECIALIST_SPECIALIZATION =
"SPCLISTSPCLIZATION";

    @PersistenceContext(unitName = "CAR_PU")
    private EntityManager em;

    @Override
    protected EntityManager getEntityManager() {
        return em;
    }

    public SpecializationFacade() {
        super(Specialization.class);
    }

    @RolesAllowed({"Reception"})
    @Override
    public void create(Specialization entity) throws AppBaseException {
        try {
            super.create(entity);
        } catch (DatabaseException e) {
            if (e.getCause() instanceof SQLNonTransientConnectionException) {

```

```

        throw
AppBaseException.createExceptionDatabaseConnectionProblem(e);
    } else {
        throw AppBaseException.createExceptionDatabaseQueryProblem(e);
    }
} catch (PersistenceException e) {
    final Throwable cause = e.getCause();
    if (cause instanceof DatabaseException &&
cause.getMessage().contains(DB_UNIQUE_CONSTRAINT_SPECIALIZATION_CODE)) {
        throw
SpecializationException.createExceptionSpecializationCodeAlreadyExists(e,
entity);
    } else {
        throw AppBaseException.createExceptionDatabaseQueryProblem(e);
    }
}
}
@RolesAllowed({"Reception"})
@Override
public void edit(Specialization entity) throws AppBaseException {
    try {
        super.edit(entity);
    } catch (DatabaseException e) {
        if (e.getCause() instanceof SQLNonTransientConnectionException) {
            throw
AppBaseException.createExceptionDatabaseConnectionProblem(e);
        } else {
            throw AppBaseException.createExceptionDatabaseQueryProblem(e);
        }
    } catch (OptimisticLockException e) {
        throw AppBaseException.createExceptionOptimisticLock(e);
    } catch (PersistenceException e) {
        throw AppBaseException.createExceptionDatabaseQueryProblem(e);
    }
} (...
@RolesAllowed({"Reception", "Patient"})
public Specialization findBySpecializationCode(String specializationCode)
throws AppBaseException {
    TypedQuery<Specialization> tq =
em.createNamedQuery("Specialization.findBySpecializationCode",
Specialization.class);
    tq.setParameter("specializationCode", specializationCode);
    try {
        return tq.getSingleResult();
    } catch (NoResultException e) {
        throw
SpecializationException.createExceptionNoSpecializationFound(e);
    } catch (PersistenceException e) {
        final Throwable cause = e.getCause();
        if (cause instanceof DatabaseException && cause.getCause()
instanceof SQLNonTransientConnectionException) {
            throw
AppBaseException.createExceptionDatabaseConnectionProblem(e);
        } else {
            throw
AppBaseException.createExceptionDatabaseQueryProblem(cause);
        }
    }
}
}

```

Listing 28: Fragment klasy SpecializationFacade.

Komponenty stanowe `@Statefull`, które reprezentują punkty dostępne [4] (ang. *endpoint*), umożliwiają komunikację warstwy prezentacji z warstwą logiki biznesowej. Odpowiadają one za transferowanie danych z formatu klas DTO [7] (ang. Data Transfer Object) na dane w formacie klas encyjných. Co dodatkowo zabezpiecza przechowywane w bazie dane. Ponadto obiekty DTO zawierają tylko informacje potrzebne dla danego przypadku. Sposób przenoszenia danych między obiektem DTO i obiektem encyjným prezentuje przykład metody `newSpecialist()` klasy `SpecialistEndpoint` widocznej na listingu 30.

3.3.2 Mechanizmy ochrony spójności danych

Mechanizm transakcji w biznesowym systemie informatycznym zapewnia spójność danych. Komponent EJB korzystając z zaimplementowanego interfejsu `javax.ejx.SessionSynchronization`, przekazuje do kontenera metody zwrotne, przedstawione na przykładzie klasy `AbstractEndpoint`, na listingu 29. Umożliwiają one zapis danych oraz identyfikację realizowanych transakcji aplikacyjnych.

Wszystkie klasy `Endpoint` posiadają adnotację `@TransactionAttribute` (`TransactionAttributeType.REQUIRES_NEW`), co zostało ukazane na przykładzie klasy `SpecialistEndpoint` we fragmencie listingu 30, Adnotacja ta skutkuje tym, że zrealizowanie metod tych klas wykonywane jest w zakresie nowo utworzonej transakcji. Dostęp do warstwy składowania danych zapewniają klasy fasad, opatrzone adnotacją `@TransactionAttribute` (`TransactionAttributeType.MANDATORY`) widoczną na listingu 28 klasy `SpecialistFacade`. Metody fasad wywoływane są tylko przez metody klas endpoint, co powoduje, że operacje te są wykonywane w zakresie transakcji zapoczątkowanej w metodzie klasy komponentu stanowego.

```
abstract public class AbstractEndpoint {
    @Resource
    SessionContext sctx;
    protected static final Logger LOGGER = Logger.getGlobal();
    private String transactionId;

    SessionSynchronization
    public void afterBegin() {
        transactionId = Long.toString(System.currentTimeMillis())
            + ThreadLocalRandom.current().nextLong(Long.MAX_VALUE);
        LOGGER.log(Level.INFO, "Transakcja TXid={0} rozpoczęta w {1},
tożsamość: {2}",
            new Object[]{transactionId, this.getClass().getName(),
sctx.getCallerPrincipal().getName()});
    }

    public void beforeCompletion() {
        LOGGER.log(Level.INFO, "Transakcja TXid={0} przed zatwierdzeniem w
{1}, tożsamość {2}",
            new Object[]{transactionId, this.getClass().getName(),
sctx.getCallerPrincipal().getName()});
    }

    public void afterCompletion(boolean committed) {
```



```

        LOGGER.log(Level.INFO, "Transakcja TXid={0} zakończona w {1} poprzez
{3}, tożsamość {2}",
            new Object[]{transactionId, this.getClass().getName(),
sctx.getCallerPrincipal().getName(),
                committed ? "ZATWIERDZENIE" : "ODWOŁANIE"});
    }
}
(...)

```

Listing 29: Klasa abstrakcyjna AbstractEndpoint

```

@Stateful
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
@Interceptors(LoggingInterceptor.class)
public class SpecialistEndpoint extends AbstractEndpoint implements
SessionSynchronization {
    @EJB
    private ReceptionFacade receptionFacade;
    @EJB
    private PatientFacade patientFacade;
    @EJB
    private AccountFacade accountFacade;
    @EJB
    private SpecializationFacade specializationFacade;
    @EJB
    private SpecialistFacade specialistFacade;
    @EJB
    private ReservationFacade reservationFacade;
    @Resource
    private SessionContext sessionContext;
    private Specialist specialistState;
    private Specialization specializationState;
    private List<Specialist> savedSpecialistStateList;
    public Specialist getSpecialistState() {
        return specialistState;
    }
    public void setSpecialistState(Specialist specialistState) {
        this.specialistState = specialistState;
    }
    private Reception loadCurrentReception() throws AppBaseException {
        String receptionLogin = sessionContext.getCallerPrincipal().getName();
        Reception receptionAccount =
receptionFacade.findByLogin(receptionLogin);
        if (receptionAccount != null) {
            return receptionAccount;
        } else {
            throw AppBaseException.createExceptionNotAuthorizedAction();
        }
    }
    public Specialist selectSpecialistWithIterator(String licenseNumber,
List<Specialist> specialists) {
        Iterator<Specialist> iterator = specialists.iterator();
        while (iterator.hasNext()) {
            Specialist specialist = iterator.next();
            if (specialist.getLicenseNumber().equals(licenseNumber)) {
                return specialist;
            }
        }
        return null;
    }
    @RolesAllowed({"Reception"})
    public void addSpecialist(SpecialistDTO specialistDTO, SpecializationDTO

```

```

specializationDTO) throws AppBaseException {
    specializationState =
specializationFacade.findBySpecializationCode(specializationDTO.getSpecializat
ionCode());
    Specialist specialist = new Specialist();
    specialist.setSpecialization(specializationState);
    specialist.setSpecialistName(specialistDTO.getSpecialistName());
    specialist.setSpecialistSurname(specialistDTO.getSpecialistSurname());
    specialist.setLicenseNumber(specialistDTO.getLicenseNumber());

    specialist.setActive(true);
    specialist.setSpecialistCreator(loadCurrentReception());

    specialistFacade.create(specialist);
}

```

Listing 30: Fragment klasy SpecialistEndpoint

Podczas równoległego przetwarzania transakcji w systemie oferującym wielodostęp, może dojść do sytuacji konfliktowej podczas zapisu i odczytu danych z bazy. Konieczne jest zastosowanie dla bazy danych izolacji transakcji. Zdefiniowana jest ona wraz z poziomem izolacji `read-committed`, pokazanej na listingu 24. Korzystając z mechanizmu blokad optymistycznych, jest możliwość sprawdzenia, czy zapisane w bazie dane nie zmieniły się od momentu ich odczytu z bazy. Kontroluje to `EntityManager` przez porównanie pola z numerem wersji, oznaczonego adnotacją `@Version`. Pole to zdefiniowane jest w klasie abstrakcyjnej `AbstractEntity` i prezentuje je listing 25. Obiekt pobrany z bazy danych jest zapamiętywany w polu stanowego komponentu EJB. Gdy wystąpi niezgodność wersji tego obiektu z wersją obiektu obecnie znajdującego się w bazie, zostanie zgłoszony widoczny na listingu 33 wyjątek `OptimisticLockException`, którego obsłużenie zapewnia metoda `edit` oraz `delete` w fasadzie, widoczna na listingu 28.

3.3.3 Kontrola dostępu

W logice biznesowej, dla poszczególnych ról ustalono kontrolę dostępu oraz uprawnienia do wykonywania określonych funkcjonalności. W rozdziale 3.4.4 Uwierzytelnianie i autoryzacja przybliżono mechanizm mapowania na role danych użytkowników dostarczonych za pośrednictwem systemu uwierzytelniania.

Korzystając z pakietu `javax.annotation.security` i adnotacji, które on oferuje, dostęp do biznesowych metod w punktach dostępowych oraz fasadach został przydzielony określonym poziomom dostępu. Na listingach klasy fasady 28 oraz 5 punktu dostępowego 30 oraz 4 przedstawiono przykład skorzystania z adnotacji `@RolesAllowed`.

3.3.4 Kontrola odpowiedzialności

W przypadku tworzenia kont użytkowników, ich edycji oraz autoryzacji, dla aktualnie istniejących w bazie danych obiektów, w systemie rejestrowana jest data wraz z godziną oraz użytkownik, który dokonał wpisu lub aktualizował informacje w bazie. Jedynie, w przypadku, gdy użytkownik sam dokonuje modyfikacji swojego konta, np. podczas zmiany własnego hasła, zamianie ulega data modyfikacji, a w kolumnie `MODIFIED_BY`,

zawierającą informacje odnośnie osoby, która ostatnio modyfikowała konto, wartość ustawiana jest na null.

Wszystkie podstawowe informacje dotyczące działania programu zapisane zostają w dziennikach zdarzeń, za co odpowiada mechanizm przechwytyjący, zaimplementowany w klasie `LoggingInterceptor` i przedstawiony na listingu 31. Metoda obiektu przechwytyjącego oznaczona jest adnotacją `@AroundInvoke`.

```
public class LoggingInterceptor {
    @Resource
    private SessionContext sessionContext;
    @AroundInvoke
    public Object additionalInvokeForMethod(InvocationContext invocation)
        throws Exception {
        StringBuilder sb = new StringBuilder("Wywołanie metody biznesowej "
            + invocation.getTarget().getClass().getName() + "."
            + invocation.getMethod().getName());
        sb.append(" z tożsamością: " +
            sessionContext.getCallerPrincipal().getName());
        try {
            Object[] parameters = invocation.getParameters();
            if (null != parameters) {
                for (Object param : parameters) {
                    if (param != null) {
                        sb.append(" z parametrem " +
                            param.getClass().getName() + "=" + param.toString());
                    } else {
                        sb.append(" z parametrem null");
                    }
                }
            }
            Object result = invocation.proceed();
            if (result != null) {
                sb.append(" zwrócono " + result.getClass().getName() + "=" +
                    result.toString());
            } else {
                sb.append(" zwrócono wartość null");
            }
            return result;
        } catch (Exception ex) {
            sb.append(" wystąpił wyjątek " + ex);
            throw ex;
        } finally {
            Logger.getGlobal().log(Level.INFO, sb.toString());
        }
    }
}
```

Listing 31: Fragment klasy `LoggingInterceptor` – mechanizm przechwytyjący

3.3.5 Obsługa błędów

W aplikacji zaimplementowany został wielopoziomowy system obsługi zgłaszanych przez nią wyjątków. Wyjątki nieobsługiwane przez aplikację [15] (tzn. wyjątki systemowe), które zgłasza kontener EJB, powodują odwołanie trwającej transakcji i wywołanie strony błędu zadeklarowanej w deskrytorze wdrożenia *web.xml*, zaprezentowanym na listingu 32 [14]. Aby zharmonizować obsługę wyjątków w aplikacji dla wszystkich wyjątków, utworzona została klasa `AppBaseException`, której fragment zaprezentowano na listingu 33. Z klasy tej dziedziczą pozostałe klasy deklarujące bardziej szczegółowe wyjątki związane z przypadkami użycia danej części aplikacji tj. `AccountException`,

`SpecialistException` i `ReservationException`. Klasa nadrzędna posiada adnotację `@ApplicationException` z atrybutem (`rollback = true`) [3], co powoduje, że mimo odwołania transakcji, komponent EJB nie zostanie zniszczony. Określone metody w kolejnych warstwach aplikacji zapewniają propagację wyjątku za pośrednictwem polecenia `throws`. Blokuje to standardowe procedury ich obsługi i powoduje, że zgłoszony wyjątek przekazywany jest do warstwy widoku aplikacji, o czym mowa w rozdziale 3.4.6 Obsługa błędów. Wówczas ma miejsce wyświetlenie odpowiedniego komunikatu użytkownikowi, informujące go o rodzaju zgłoszonego wyjątku i dalszych instrukcjach.

```
(...)
<error-page>
  <error-code>403</error-code>
  <location>/faces/error/error403.xhtml</location>
</error-page>
<error-page>
  <error-code>404</error-code>
  <location>/faces/error/error404.xhtml</location>
</error-page>
<error-page>
  <error-code>500</error-code>
  <location>/faces/error/error500.xhtml</location>
</error-page>
<error-page>
  <exception-type>java.lang.RuntimeException</exception-type>
  <location>/faces/error/error.xhtml</location>
</error-page> (...)
```

Listing 32: Fragment pliku konfiguracyjnego `web.xml` - deskryptora wdrożenia aplikacji

```
@ApplicationException(rollback = true)
public class AppBaseException extends Exception {
    static final public String KEY_OPTIMISTIC_LOCK =
"error.optimistic.lock.problem";
    static final public String KEY_REPEATED_TRANSACTION_ROLLBACK =
"error.repeated.transaction.rollback.problem";
    static final public String KEY_DATABASE_QUERY_PROBLEM =
"error.database.query.problem";
    static final public String KEY_DATABASE_CONNECTION_PROBLEM =
"error.database.connection.problem";
    static final public String KEY_NOT_AUTHORIZED_ACTION =
"error.not.authorized.account.problem";
    static final public String KEY_NO_RESULT = "error.no.result.problem";
    public AppBaseException() {
    }
    protected AppBaseException(String message) {
        super(message);
    }
    protected AppBaseException(String message, Throwable cause) {
        super(message, cause);
    }
    static public AppBaseException
createExceptionForRepeatedTransactionRollback() {
        return new AppBaseException(KEY_REPEATED_TRANSACTION_ROLLBACK);
    }
    public static AppBaseException
createExceptionDatabaseQueryProblem(Throwable e) {
        return new AppBaseException(KEY_DATABASE_QUERY_PROBLEM);
    }
}
```

```

    }
    public static AppBaseException
createExceptionDatabaseConnectionProblem(Throwable e) {
        return new AppBaseException(KEY_DATABASE_CONNECTION_PROBLEM);
    }
    public static AppBaseException
createExceptionOptimisticLock(OptimisticLockException e) {
        return new AppBaseException(KEY_OPTIMISTIC_LOCK);
    }
    public static AppBaseException createExceptionNotAuthorizedAction() {
        return new AppBaseException(KEY_NOT_AUTHORIZED_ACTION);
    }
    public static AppBaseException createExceptionNoResult(NoResultException
e) {
        return new AppBaseException(KEY_NO_RESULT, e);
    }
}

```

Listing 33: Fragment klasy wyjątków AppBaseException.

Obsługa błędów w punktach dostępowych polega na deklarowaniu własnych wyjątków aplikacyjnych i wywoływaniu ich w sytuacjach gdy przekazywane przez użytkownika aplikacji dane nie spełniają określonych przez nas kryteriów, lub też gdy wystąpiły inne warunki jak np. jednoczesne edytowanie tych samych danych przez wielu użytkowników (wielodostęp).

Obsługę takich wyjątków prezentuje listing 34 metody addReservation(), odpowiedzialnej wprowadzanie do bazy danych nowej rezerwacji.

```

@RolesAllowed({"Patient"})
public void addReservation(ReservationDTO reservationDTO, SpecialistDTO
specialistDTO, SpecializationDTO specializationDTO) throws AppBaseException {

    LocalDateTime selectedDateTime =
(reservationDTO.getReservationDate()).toInstant()
.atZone(ZoneId.systemDefault()).toLocalDateTime();
    LocalDate selectedDate = selectedDateTime.toLocalDate();

    if (selectedDateTime.isBefore(LocalDateTime.now())) {
        ContextUtils.emitI18NMessage("AddReservationForm3:reservationDate",
"error.add.reservation3.past.date");
        throw ReservationException.createExceptionDateTimeProblemPastDate();
    }

    // Święta stałe ustawowo wolne od pracy
    List<LocalDate> permanentHolidaysList = new ArrayList<>();
    permanentHolidaysList.add(LocalDate.of(2020, Month.JANUARY, 1));
    permanentHolidaysList.add(LocalDate.of(2020, Month.JANUARY, 6));
    permanentHolidaysList.add(LocalDate.of(2020, Month.MAY, 1));
    permanentHolidaysList.add(LocalDate.of(2020, Month.MAY, 3));
    permanentHolidaysList.add(LocalDate.of(2020, Month.AUGUST, 15));
    permanentHolidaysList.add(LocalDate.of(2020, Month.NOVEMBER, 1));
    permanentHolidaysList.add(LocalDate.of(2020, Month.NOVEMBER, 11));
    permanentHolidaysList.add(LocalDate.of(2020, Month.DECEMBER, 25));
    permanentHolidaysList.add(LocalDate.of(2020, Month.DECEMBER, 26));

    // Święta ruchome ustawowo wolne od pracy
    List<LocalDate> moveableHolidaysList = new ArrayList<>();
    moveableHolidaysList.add(LocalDate.of(2020, Month.APRIL, 12));
    moveableHolidaysList.add(LocalDate.of(2020, Month.APRIL, 13));
    moveableHolidaysList.add(LocalDate.of(2020, Month.MAY, 31));
    moveableHolidaysList.add(LocalDate.of(2020, Month.JUNE, 11));

    for (LocalDate permanentHoliday : permanentHolidaysList) {
        if ((permanentHoliday.getMonth().equals(selectedDate.getMonth())) &&

```

```

(permanentHoliday.getDayOfMonth() == selectedDate.getDayOfMonth())) {
    ContextUtils.emitI18NMessage("AddReservationForm3:reservationDate",
"error.add.reservation3.holiday");
    throw ReservationException.createExceptionDateTimeProblemHoliday();
}

if (moveableHolidaysList.contains(selectedDate)) {
    ContextUtils.emitI18NMessage("AddReservationForm3:reservationDate",
"error.add.reservation3.holiday");
    throw ReservationException.createExceptionDateTimeProblemHoliday();
}

if (selectedDate.getDayOfWeek().equals(DayOfWeek.SATURDAY) ||
selectedDate.getDayOfWeek().equals(DayOfWeek.SUNDAY)) {
    ContextUtils.emitI18NMessage("AddReservationForm3:reservationDate",
"error.add.reservation3.weekend");
    throw ReservationException.createExceptionDateTimeProblemWeekend();
}

if (selectedDate.isBefore(LocalDate.now().plusDays(1))) {
    ContextUtils.emitI18NMessage("AddReservationForm3:reservationDate",
"error.add.reservation3.next.day.restriction");
    throw
ReservationException.createExceptionDateTimeProblemNextDayRestriction();
}
if (selectedDate.isAfter(LocalDate.now().plusMonths(3))) {
    ContextUtils.emitI18NMessage("AddReservationForm3:reservationDate",
"error.add.reservation3.three.months.restriction.");
    throw
ReservationException.createExceptionDateTimeProblemMaxThreeMonthsRestriction();
}
if (selectedDate.equals(LocalDate.of(2019, Month.NOVEMBER, 25))) {
    ContextUtils.emitI18NMessage("AddReservationForm3:reservationDate",
"error.houston.we.have.a.problem");
    throw
ReservationException.createExceptionDateTimeEasterEggBirthdaySurprise();
}

if (selectedDateTime.getHour() < 8 || selectedDateTime.getHour() > 17) {
    ContextUtils.emitI18NMessage("AddReservationForm3:reservationDate",
"error.add.reservation3.off.hours");
    throw ReservationException.createExceptionDateTimeProblemOffHours();
}

specialistState =
specialistFacade.findByLicenseNumber(specialistDTO.getLicenseNumber());
specializationState =
specializationFacade.findBySpecializationCode(specializationDTO
.getSpecializationCode());
if (!specializationState.getSpecializationName().equals(specialistDTO
.getSpecialization().getSpecializationName())) {
    throw SpecializationException
.createExceptionSpecializationEdited(specializationState);
}

if (specialistState.isActive()) {
    if (specialistState.getSpecialistName().equals(specialistDTO
.getSpecialistName()) &&
specialistState.getSpecialistSurname().equals(specialistDTO
.getSpecialistSurname())) {

        Reservation reservation = new Reservation();
        reservation.setReservedSpecialist(specialistState);
        reservation.setReservationDate(reservationDTO.getReservationDate());
        reservation.setReservationCreator(loadCurrentPatient());
        reservationFacade.create(reservation);
    } else {
        throw
SpecialistException.createExceptionEditedSpecialist(specialistState);
    }
}

```

```

    }
    } else {
        throw
        SpecialistException.createExceptionInactiveSpecialist(specialistState);
    }
}

```

Listing 34: Fragment klasy ReservationEndpoint metoda addReservation()

3.4 Warstwa widoku

Warstwa widoku w aplikacji oparta została głównie na implementacji szkieletu JSF (ang. *JavaServer Faces*) [14] oraz języku HTML [1], które obecne są we wszystkich jej stronach. Aby widok stron został ujednolicony, wykorzystano biblioteki kaskadowych arkuszy stylów CSS (ang. *Cascading Style Sheets*) zapewnianych przez Bootstrap [2], co pozwoliło na podział widoku na kilka obszarów: nagłówek, pasek menu, zawartość główna, oraz stopka. Bardziej zaawansowane funkcje, przykładowo, interaktywny kalendarz, czy też potwierdzenie usunięcia danych z tabeli, zostały zaimplementowane z wykorzystaniem języka JavaScript [16].

3.4.1 Wzorzec projektowy DTO

W systemie został wykorzystany model danych DTO [6], Jest to Obiekt Transferu Danych (ang. *Data Transfer Object*), który służy do odseparowania od siebie danych przekazywanych między warstwą składowania danych, a warstwą prezentacji. DTO przenosi dane między warstwą widoku i warstwą logiki biznesowej, gdzie wewnątrz klas punktu dostępowego współdziała z obiektami klas encyjnymi i transferuje między nimi wybrane dane. Poprzez takie rozwiązanie, użytkownik aplikacji nigdy nie ma dostępu do krytycznych dla aplikacji danych, takich jak `Version` lub `ID`, jak również do danych nieistotnych dla obsługi interfejsu użytkownika, a ważnych ze względu na bezpieczeństwo, jak np. `password` lub dane, które dotyczą kontroli odpowiedzialności np. `accountCreator` lub `modifiedBy`. Przykładem klasy DTO jest `SpecialistDTO`, której fragment prezentuje listing 35.

```

public class SpecialistDTO implements Comparable<SpecialistDTO> {

    private Specialization specialization;
    private String specialistName;
    private String specialistSurname;
    private String licenseNumber;
    private boolean active;
    private boolean presentInReservation;
    public SpecialistDTO(Specialization specialization, String specialistName,
String specialistSurname, String licenseNumber) {
        this.specialization = specialization;
        this.specialistName = specialistName;
        this.specialistSurname = specialistSurname;
        this.licenseNumber = licenseNumber;
    }
    (...)
    @Override
    public String toString() {
        return specialistName + " " + specialistSurname + ", " +
specialization;
    }
}

```

```

    }

    @Override
    public int compareTo(SpecialistDTO o) {
        return this.specialistSurname.compareTo(o.specialistSurname);
    }

```

Listing 35: Fragment klasy SpecialistDTO

Ponadto klasy DTO korzystają z implementacji interfejsu `Comparable`, który poprzez metodę `compareTo` przedstawionej na listingu 35, umożliwia porównywanie obiektów po jego pewnych cechach i zwracania ich w postaci posortowanej listy [15]. Sortowanie list odbywa się z pomocą metody `sort` klasy `Collections` co zaprezentowano na przykładzie listingu 9 klasy punktu dostępowego *Endpoint*.

3.4.2 Ujednolicony interfejs użytkownika

Interfejs użytkownika zrealizowany został przy użyciu stron www dynamicznie generowanych, których wygląd został ujednolicony. Jest on niezależny od poziomu dostępu, jaki posiada dany użytkownik. Wszystkie strony aplikacji posiadają ujednolicony interfejs graficzny niezależnie od poziomu dostępu użytkownika i zostały wykonane przy pomocy technologii JSF, HTML i CSS, które zapewniają odpowiednie przystosowanie wyglądu stron poprzez oferowane biblioteki.

Nawigacja pomiędzy stronami odbywa się dzięki innym dla każdego poziomu dostępu przyciskom na pasku menu, zaprezentowanym na rysunkach 39, 40, 41 i 42. Aplikacja wykorzystuje też nawigację dostępną poprzez polecenia `<navigation-case>`, które znajdują się w pliku konfiguracyjnym *faces-config.xml*, którego fragment pokazano na listingu 39. Znajdujące się tam reguły nawigacji są wykorzystywane w klasach typu `PageBean` w metodach klasy `String`, które w zwracanej wartości zawierają daną regułę nawigacji.

JSF dostarcza liczne wzorce szablonów JSF. Listing 36, przedstawia kod szablonu, który został wykorzystany do podzielenia wszystkich stron warstwy widoku na osobne sekcje, w zależności od przydzielonego użytkownikowi poziomu dostępu. Zastosowane w nim polecenie `<ui:insert name=" " >` definiuje każdą nową sekcję.

```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:p="http://primefaces.org/ui">
    <h:head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <h:outputStylesheet name="/css/bootstrap.min.css"/>
        <h:outputStylesheet name="/css/font-awesome.min.css"/>
        <h:outputStylesheet name="/css/style.css"/>
        <h:outputScript name="/js/jquery.js"/>
        <h:outputScript name="/js/app.js"/>

```



```

        <h:outputScript name="/js/bootstrap.min.js"/>
        <h:outputStylesheet name="/css/bootstrap-datetimepicker-
standalone.min.css" />
        <h:outputStylesheet name="/css/select2.min.css" />
        <h:outputStylesheet name="/css/bootstrap-datetimepicker.min.css" />
        <title>#{msg['page.main.template.title']}

```

Listing 36: Fragment szablonu mainTemplate.xhtml

Nagłówek (header) widoczny jest na rysunku 38, pasek menu (menu) przedstawiono na rysunkach 39, 40, 41 i 42, zawartość główną (content) oraz stopkę (footer) zaprezentowano na rysunku 43. Poprzez zastosowanie polecenia `<ui:composition template="./mainTemplate.xhtml">` strony, które korzystają z szablonu mogą implementować całą domyślną zawartość sekcji szablonu i wprowadzać zmiany tylko w pożądanym sekcjach, co zaprezentowano na listingu 37, który jest fragmentem kodu strony `mainPage.xhtml`, który zmienia jedynie sekcję `<ui:define name="content">`.

```

    (...)<body>
        <ui:composition template="./mainTemplate.xhtml">
            <ui:define name="content">
                <h:messages globalOnly="true" styleClass="error_large" />
                <h:messages id="success" for="success"
styleClass="confirm_large" />
                #{msg['page.main.page.content.welcome']}
            </ui:define>
        </ui:composition>
    </body>
</html>

```

Listing 37: Fragment strony mainPage.xhtml



Rysunek 38: Wygląd sekcji nagłówka strony.

Rysunek 39: Wygląd paska menu dla użytkownika nieuwierzytelnionego.

Rysunek 40: Wygląd paska menu dla poziomu dostępu: Administrator

Rysunek 41: Wygląd paska menu dla poziomu dostępu: Recepcja

Rysunek 42: Wygląd paska menu dla poziomu dostępu: Pacjent

Konsultacje specjalistyczne z zakresu:

- ortopedii
- traumatologii
- diagnostyki USG
- chirurgii ogólnej
- kardiologii, w tym kardiologii sportowej
- neurologii, badania EMG
- neurochirurgii
- podologii ortopedycznej
- flebologii
- reumatologii
- kardiochirurgii
- badań okresowych sportowców
- dietetyki
- psychologii

Prowadzenie rehabilitacji:

- pourazowej
- po zabiegach operacyjnych
- przed planowanym zabiegiem operacyjnym

Rysunek 43: Wygląd sekcji zawartości głównej na przykładzie strony mainPage.xhtml i sekcji stopki.

3.4.3 Internacjonalizacja

Aplikacja została przygotowana do działania z dwiema wersjami językowymi: polskiej i angielskiej. W interfejsie użytkownika zaimplementowano internacjonalizację z wykorzystaniem standardu i18n. Obsługa wersji językowej jest automatycznie ładowana poprzez ustawienia przeglądarki internetowej użytkownika systemu. Aby możliwe było korzystanie z internacjonalizacji niezbędna jest jej konfiguracja w pliku deskryptora wdrożenia web.xml pokazana we fragmencie listingu 51. Deskryptor wdrożenia przechowuje informację o parametrze *resourceBundle.path*.

```
(...) <context-param>
    <param-name>resourceBundle.path</param-name>
    <param-value>i18n.messages</param-value>
</context-param>
```

Listing 38: Fragment pliku deskryptora wdrożenia web.xml odpowiedzialny za konfigurację zasobu dla internacjonalizacji

Wartość *i18n.messages* z listingu oznacza, że pliki właściwości z zawartością przechowywanych par klucz.właściwość=wartość.właściwość znajdują się w plikach *messages.properties*, te z kolei umieszczone są w domyślnej lokalizacji *|src|main|resources|messages*. Pliki z internacjonalizacją dla poszczególnych języków, muszą kończyć się dla języka polskiego oznaczeniem „_pl” i adekwatnie „_en” dla języka angielskiego.

Plik *faces_config.xml* przedstawiony na listingu 39 zawiera konfigurację lokalizacji, gdzie wspierane przez aplikację lokalizacje to pl i en, a domyślną lokalizacją jest pl. W katalogu i18n w pliku *messegas* znajdują się komunikaty generowane w celu komunikowania się z użytkownikiem poprzez parametr *msg*, którego fragment obrazuje rysunek nr 52

```
(...) <locale-config>
    <default-locale>pl</default-locale>
    <supported-locale>pl</supported-locale>
    <supported-locale>en</supported-locale>
</locale-config>

<resource-bundle>
    <base-name>i18n.messages</base-name>
    <var>msg</var>
</resource-bundle>
<message-bundle>i18n.jsf_messages</message-bundle>
<navigation-case>
    <from-outcome>addReservation2</from-outcome>
    <to-view-id>/addReservation2.xhtml</to-view-id>
    <redirect />
</navigation-case>
```

Listing 39: Fragment pliku faces_config.xml

```

error.not.authorized.account.problem= Brak odpowiedniego poziomu dostępu do wykonania operacji.
error.account.not.active.problem= Twoje konto jest nieaktywne. Skontaktuj się z administratorem.
error.wrong.access.level= Wymagany poziom dostępu:
error.account.already.active.problem= Konto było już aktywowane.
error.account.already.deactive.problem= Konto było już deaktywowane.
error.account.already.changed.problem= Konto było edytowane przez innego administratora.

error.specialist.license.number.exist.problem= Specjalista o podanym numerze PWZ już istnieje.
error.specialist.wrong.state.problem= Wykryto niezgodność edytowanego specjalisty.
error.specialist.is.inactive.problem = Wybrany specjalista jest niedostępny.
error.specialist.was.edited.problem= Dane specjalisty zostały zmienione, spróbuj ponownie.

error.specialist.already.reserved.problem= Nie można usunąć specjalisty, ponieważ jest on przypisany do rezerwacji.
specialist.cannot.be.empty = Proszę najpierw wybrać specjalistę.

error.patient.and.date.exist.problem= Pacjent posiada już wizytę w podanym terminie.
error.specialist.and.date.exist.problem= Specjalista nie jest dostępny w podanym terminie.

error.specialization.specializationCode.exist.problem = Podany kod specjalizacji został już użyty.
error.select.specialization.problem= Proszę najpierw wybrać specjalizację.

```

Rysunek 44: Fragment pliku `messages_pl.properties`.

Aby skrót komunikatu wyświetlił wartość, która jest odpowiednia dla wybranego języka, niezbędne jest jego przekonwertowanie przez zadeklarowanie go na stronach JSF wewnątrz specjalnego polecenia, np. `{msg[page.register.title]}`. Wynikiem tego jest wywołanie metod klasy `ContextUtils`, której fragment zaprezentowano na listingu 40. Metoda ta zwraca użytkownikowi właściwy komunikat.

```

(...)public static void emitI18NMessage(final String id, final String key) {
    FacesMessage msg = new FacesMessage(getI18NMessage(key));
    FacesContext.getCurrentInstance().addMessage(id, msg);
}
public static String printI18NMessage(final String key) {
    String msg = getI18NMessage(key);
    return msg; }

```

Listing 40: Fragmenty klasy `ContextUtils`.

3.4.4 Uwierzytelnianie i autoryzacja

Aby dokonać uwierzytelnienia się w systemie, konieczne jest posiadanie przez użytkownika unikalnego loginu oraz hasła. Należy je umieścić w formularzu zaprezentowanym na rysunku 6, który korzysta z poleceń widocznych na listingu 41 dotyczącego strony uwierzytelniania `login.xhtml`. Konto, do którego są przypisane te właściwości musi być aktywne. Po udanym zalogowaniu użytkownik zostaje przeniesiony na stronę główną `mainPage.xhtml`, co zostało zadeklarowane w pliku konfiguracyjnym `web.xml`, którego fragment widzimy na listingu 42.

```

<form method="post" action="j_security_check">(…)
<input type="text" name="j_username" />(…)
<input type="password" name="j_password" />(…)
<input type="submit" value="" />(…)

```

Listing 41: Fragment pliku `login.xhtml`

```

<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>CAR_JDBC_Realm</realm-name>
</form-login-config>

```

```

        <form-login-page>/faces/login.xhtml</form-login-page>
        <form-error-page>/faces/errorAuthentication.xhtml</form-error-
page>
    </form-login-config>
</login-config>

```

Listing 42: Fragment deskryptora wdrożenia web.xml

Każda z ról zaimplementowanych w aplikacji posiada dostęp do innych funkcji, co możliwe było dzięki zastosowaniu znacznika `<f:subview>`, w elementach paska menu, co pokazują rysunki 39, 40, 41 i 42, fragment klasy *mainTemplate.xhtml* zawierający podział widoku został przedstawiony na listingu 43.

```

<f:subview id="patientView" rendered="#{request.isUserInRole('Patient')}}">
    <li><a href="newReservation.xhtml">(...)
    <li><a href="listMyCurrentReservations.xhtml">(...)
</f:subview> (...)
<f:subview id="receptionView" rendered="#{request.isUserInRole('Reception')}}">
    <li><a href="listCurrentReservations.xhtml"> (...)
</f:subview>
<f:subview id="unauthenticatedView" rendered="#{empty request.remoteUser}">
    <li><a href="resetPassword.xhtml">(...)

```

Listing 43: Fragment pliku mainTemplate.xhtml przedstawiający podział w widoku menu

Fragment deskryptora wdrożenia aplikacji *web.xml* został przedstawione na listingu 44. Odpowiada on za deklarację ról i ograniczeń dostępu do wybranych zasobów aplikacji, wraz z ustawieniami wykorzystania protokołu HTTPS [1] (ang. *Hypertext Transfer Protocol Secure*), szyfrującego dane przy pomocy protokołu SSL [11] (ang. *Secure Socket Layer*).

```

<security-constraint>
<display-name>AuthenticatedUserPages</display-name>
<web-resource-collection>
    <web-resource-name>AuthenticatedUserPages</web-resource-name>
    <description/>
    <url-pattern>/faces/displayMyAccount.xhtml</url-pattern>
    <url-pattern>/faces/editMyAccount.xhtml</url-pattern>
    <url-pattern>/faces/changeMyPassword.xhtml</url-pattern>
</web-resource-collection>
    <auth-constraint>
    <description/>
    <role-name>Admin</role-name>
    <role-name>Reception</role-name>
    <role-name>Patient</role-name>
</auth-constraint>
    <user-data-constraint>
    <description/>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
</security-constraint> (...)
<security-role>
    <description/>
    <role-name>Admin</role-name>
</security-role>
<security-role>
    <description/>
    <role-name>Reception</role-name>
</security-role>

```

```

<security-role>
    <description/>
    <role-name>Patient</role-name>
</security-role>

```

Listing 44: Fragmenty kodu deskryptora wdrożenia web.xml

Deskryptor wdrożenia aplikacji *glassfish-web.xml*, którego fragment jest widoczny na listingu 45 jest odpowiedzialny za mapowanie użytkowników i grup na odpowiadające im role.

```

<context-root>CAR</context-root>
<security-role-mapping>
    <role-name>Admin</role-name>
    <group-name>access.level.admin</group-name>
</security-role-mapping>
<security-role-mapping>
    <role-name>Reception</role-name>
    <group-name>access.level.reception</group-name>
</security-role-mapping>
<security-role-mapping>
    <role-name>Patient</role-name>
    <group-name>access.level.patient</group-name>
</security-role-mapping>

```

Listing 45: Fragmenty deskryptora wdrożenia aplikacji glassfish-web.xml

Uwierzytelnianie w aplikacji odbywa się poprzez serwer aplikacyjny Payara, który zapewnia tzw. obszar uwierzytelniania (ang. *security realm*), w którym należy podać nazwę zadeklarowaną wcześniej w deskrytorze wdrożenia, widoczną na listingu 42. Konfiguracja serwera została pokazana w rozdziale 3.5.4 Konfiguracja obszaru bezpieczeństwa na serwerze.

Hasła wszystkich użytkowników które wymagane są do uwierzytelnienia się na serwerze aplikacji, przechowywane są w bazie danych w postaci niejawnej i są skrótem zaszyfrowanym algorytmem SHA-256. Szyfrowanie hasła odbywa się w widocznej na listingu 46 metodzie `setPassword()` klasy encyjnej `Account`.

```

public void setPassword(String password) {
    try {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] encodedhash =
        digest.digest(password.getBytes(StandardCharsets.UTF_8));
        StringBuffer stringBuffer = new StringBuffer();
        for (int i = 0; i < encodedhash.length; i++) {
            stringBuffer.append(Integer.toString((encodedhash[i] & 0xff) +
            0x100, 16).substring(1)); }
        this.password = stringBuffer.toString();
    } catch (NoSuchAlgorithmException ex) {
        Logger.getLogger(Account.class.getName()).log(Level.SEVERE, null,
        ex); }
}

```

Listing 46: Fragment klasy encyjnej Account przedstawiający metodę setPassword()

3.4.5 Walidacja danych

Walidacja danych zapobiega wprowadzeniu do formularzy niespójnych danych, które nie spełniają określonych formatów i reguł biznesowych. Proces ten odbywa się z wykorzystaniem walidatorów, które umieszczone są wewnątrz plików xhtml w warstwie

widoku projektu, co pokazuje listing przedstawiający walidację danych dodawanego specjalisty 47. Walidatory są wyrażeniami regularnymi i kontrolują format wprowadzanych przez użytkownika danych, tak aby był zgodny z formatem wymagany przy wprowadzaniu ich do bazy danych.

```
(...)<h:inputText id="surname" maxLength="100" required="true"
validatorMessage="#{msg['page.add.specialist.validator.specialist.surname']}"
value="${addSpecialistPageBean.specialistDTO.specialistSurname}" >
    <f:validateLength minimum="3" maximum="100" ></f:validateLength>
    <f:validateRegex pattern="[A-ZĄĆĘŁŃÓŚŻ] [a-ząćęłńóśż]*([\s\-\'] [A-
ZĄĆĘŁŃÓŚŻ] [a-ząćęłńóśż])*" />
</h:inputText>
<h:message for="surname" styleClass="error_small"/>
<label class="text-left">
    <h:outputLabel value="#{msg['page.specialist.label.license.number']}" />
```

Listing 47: Fragment pliku addSpecialist.xhtml – walidacja wprowadzanych danych.

W przypadku niepoprawnego wprowadzenia danych, walidator wyświetla odpowiedni, internacjonalizowany komunikat błędu, widoczny na listingu 47 jako wartość parametru `validatorMessage`.

3.4.6 Obsługa błędów

W warstwie widoku, obsługa błędów następuje w klasach typu `PageBean`, poprzez obsługę propagowanych do niej wyjątków aplikacyjnych [12]. Odbywa się to poprzez użycie deklaracji `throws` występującej we wszystkich metodach pozostałych klas, co zaprezentowane jest na listingach 28 lub 30. Klasa `PageBean` może również deklarować swoje własne wyjątki w formie komunikatów podlegających internacjonalizacji i wyświetlać je razem z wyjątkami aplikacyjnymi, przy użyciu klasy `ContextUtils`. Odpowiednim przykładem będzie w tym miejscu metoda `registerAccountAction()`, widoczna na listingu 48, która zawarta jest w klasie `newAccountPageBean`. Komunikaty te wyświetlają się na stronach JSF dzięki użyciu elementów `<h:messages(...)/>` przedstawionych na przykładzie strony `newAccount.xhtml` na listingu 49. Poza tym w warstwie widoku wywoływany jest `Logger`, który odnotowuje wystąpienie wyjątku w dzienniku zdarzeń, a także zgłasza jego poziom [10].

```
(...) public String registerAccountAction() {
    if (passwordRepeat.equals(accountDTO.getPassword())) {
        try {
            accountControllerBean.registerAccount(accountDTO);
        } catch (AppBaseException ex) {
            Logger.getLogger(NewAccountPageBean.class.getName()).log(Level.SEVERE,
            null, ex);

            ContextUtils.emitI18NMessage(ex.getMessage().equals("error.account.login.exist
            s.problem") ? "RegisterForm:login" : null, ex.getMessage());
            return null;
        } else {
            ContextUtils.emitI18NMessage("RegisterForm:passwordRepeat",
            "passwords.not.matching");
            return null;
        }
        return "main"; } (...)
```

Listing 48: Fragment klasy NewAccountPageBean.

```

<label class="text-left">
    <h:outputLabel value="#{msg['page.register.form.label.name']}" />
</label>
<h:inputText class="margin-bottom-small" id="name" maxlength="60"
required="true" validatorMessage="#{msg['page.account.validator.name']}"
    value="${accountRegistrationPageBean.accountDTO.name}" >
    <f:validateLength minimum="2" maximum="60" ></f:validateLength>
    <f:validateRegex pattern="[A-ZĄĆĘŁŃÓŚŻ] [a-ząćęłńóśż]+([\s][A-ZĄĆĘŁŃÓŚŻ]
[a-ząćęłńóśż]+)?" />
</h:inputText>
<h:message for="name" styleClass="error_small"/>
<label class="text-left">
    <h:outputLabel value="#{msg['page.register.form.label.surname']}" />
</label>
<h:inputText class="margin-bottom-small" id="surname" maxlength="100"
required="true" validatorMessage="#{msg['page.account.validator.surname']}"
    value="${accountRegistrationPageBean.accountDTO.surname}" >
    <f:validateLength minimum="2" maximum="100"></f:validateLength>
    <f:validateRegex pattern="[A-ZĄĆĘŁŃÓŚŻ] [a-ząćęłńóśż]*([\s\-\'] [A-
ZĄĆĘŁŃÓŚŻ] [a-ząćęłńóśż]*)*" />
</h:inputText>
<h:message for="surname" styleClass="error_small"/>

```

Listing 49: Fragment strony registerAccount.xhtml

3.4.7 Technologie do obsługi interfejsu graficznego

Najistotniejszą technologią użytą do przygotowania interfejsu graficznego jest szkielet JSF, którego wykorzystanie zostało dokładniej opisane w rozdziale 3.4.2 Ujednolicony interfejs użytkownika. Utworzony dzięki JSF szablon *mainTemplate.xhtml* przedstawiony na listingu 36 oraz korzystające z niego strony klientów szablonu, z listingów 1, 7, 13 i 18 pozwalają utrzymać jednolity wygląd systemu i ułatwiają rozbudowywanie go o kolejne elementy.

Warstwa widoku korzysta też z biblioteki CSS *Bootstrap*, której użycie pozwoliło na utworzenie responsywnego paska menu, widocznego na rysunkach 39, 40, 41 i 42. Pasek ten, po zmianie rozdzielczości lub rozmiaru okna strony odpowiednio dostosowuje swój wygląd. Fragment kodu strony *mainTemplate.xhtml* odpowiedzialny na tworzenie paska menu znajduje się na listingu 50.

```

<div class="row">
    <div class="col-md-12 menu">
        <ui:insert name="menu">
            <div class="row">
                <nav class="navbar navbar-inverse">
                    <div class="container-fluid">
                        <div class="navbar-header">
                            <button type="button" class="navbar-toggle" data-
toggle="collapse" data-target="#myNavbar">
                                <span class="icon-bar"></span>
                                <span class="icon-bar"></span>
                                <span class="icon-bar"></span>
                            </button>
                            <a class="navbar-
brand">#{msg['page.main.template.label.menu']}</a>
                        </div>
                        <div class="collapse navbar-collapse" id="myNavbar">
                            <ul class="nav navbar-nav">

```

Listing 50: Fragment szablonu mainTemplate.xhtml odpowiedzialny za wyświetlenie paska menu

Aby wyświetlić przedstawiony na rysunku 45 interaktywny kalendarz dla przypadku użycia związanego z tworzeniem nowej rezerwacji wizyty, wykorzystano bibliotekę języka programowania *JavaScript* o nazwie *Date Range Picker*. Jej użycie deklarują polecenia `<h:outputStylesheet (...)/>` i `<h:outputScript (...)/>` przedstawione na listingu 36 strony *mainTemaplete.xhtml*., a także polecenie `<h:inputText class="datetimepicker (...)/>` strony *newReservation.xhtml* widocznej na listingu 51.

Pusta lista rozwijana oznacza chwilowy brak dostępnych specjalistów lub specjalizacji.

Rysunek 45: Strona newReservation.xhtml - wybór zakresu dat z kalendarza

```
<h:inputText class="datetimepicker" id="reservationDate" maxlength="13"
size="13" required="true"
validatorMessage="page.reservation.validator.reservationtime"
value="\${addReservationPageBean3.reservationDTO.reservationDate}" >
<f:convertDateTime type="both" pattern="yyyy-MM-dd HH"/>
</h:inputText>
<h:message for="reservationDate" styleClass="error_small" />
```

Listing 51: Fragment strony newReservation3.xhtml

Zastosowanie ziaren CDI [8] (ang. *Contexts and Dependency Injection*), czyli klas, których cykl życia realizowany jest przez kontener EJB, zostało użyte do stworzenia warstwy widoku. Cykl życia ziaren CDI zależy od funkcji, jaką pełni dana klasa w aplikacji. Ziarna te obsługują żądania użytkownika aplikacji i przekazują określone dane biznesowe na strony JSF. W aplikacji wykorzystano ziarna o zasięgu [14]:

- żądania `@RequestScoped` dla większości stron, w których występują formularze, jak na listingach 48 i 2

- sesji `@SessionScoped` w klasach kontrolera, które odpowiadają za zapamiętywanie i przekazywanie danych do następnych żądań oraz komunikacji z warstwą logiki biznesowej pokazanych na listingu 3 oraz 20.
- widoku `@ViewScoped` w klasach odpowiedzialnych za wyświetlanie listy, której przykład przedstawiono na listingu 8 i 14. Ważne jest tu utrzymanie stanu wyświetlonej listy, aby nie pozwolić na wybranie błędnych danych z uwagi na wielodostęp;
- aplikacyjnym `@ApplicationScoped` dla klasy `MainApplicationBean`, zaprezentowanej na listingu 52. Klasa ta zawiera metody wyświetlające login oraz poziom dostępu użytkownika na pasku menu,

```
@Named(value = "mainApplicationPageBean")
@ApplicationScoped
public class MainApplicationBean {
    (...)
    public String logOutAction() {
        ContextUtils.invalidateSession();
        return "main";
    }
    public String getMyLogin() {
        return ContextUtils.getUserName();
    }
    public String myAccessLevel() {
        return accountEndpoint.getI18nAccountForAccessLevelDisplay();
    }
    (...)
    return null; }
}
```

Listing 52: Fragment klasy MainApplicationPageBean

3.5 Instrukcja wdrożenia

Do uruchomienia stworzonego systemu niezbędne jest przygotowanie środowiska, które posiada zainstalowane:

- system operacyjny,
- współczesną przeglądarkę internetową,
- uruchomiony serwer aplikacyjny Payara w wersji 5.183,
- uruchomiony system Java DB (*Apache Derby*) w wersji 10.13.1.1.

3.5.1 Utworzenie bazy danych

Utworzenie bazy danych jest możliwe za pomocą programu narzędziowego iJ dostarczonego wraz z *Apache Derby*.

Listing 24 przedstawia konfigurację bazy danych w pliku *glassfish-resources.xml*, natomiast sam proces tworzenia bazy zaprezentowano w listingu 25.

Bazę danych o nazwie `jdbc:derby://localhost:1527/CAR` należy utworzyć w katalogu z bazami danych zlokalizowanym w katalogu domowym używając polecenia `connect`, a następnie ustawić wartość `create` na `true`, jak zaprezentowano na listingu 53, dla domyślnego dla Derby katalogu `.netbeans-derby`.

```

<jdbc-resource enabled="true" jndi-name="jdbc/CAR_DS" object-type="user"
  pool-name="CAR_CP"> (...) </jdbc-resource>
<jdbc-connection-pool transaction-isolation-level="read-committed" (...)
  <property name="URL" value="jdbc:derby://localhost:1527/CAR"/>
  <property name="serverName" value="localhost"/>
  <property name="PortNumber" value="1527"/>
  <property name="DatabaseName" value="CAR"/>
  <property name="User" value="cliniccar"/>
  <property name="Password" value="cliniccar"/>
</jdbc-connection-pool></resources>

```

Listing 53: Fragment pliku konfiguracyjnego glassfish-resource.xml

Bazę danych o nazwie:

`jdbc:derby://localhost:1527/CAR` należy utworzyć w katalogu z bazami danych zlokalizowanym w katalogu domowym używając polecenia `connect`, następnie ustawić `create` na `true`, jak zaprezentowano na listingu 54, dla domyślnego dla Derby katalogu `.netbeans-derby`.

3.5.2 Umieszczenie w bazie struktur bazy danych i danych inicjujących

Następnie należy, zgodnie z poleceniami pokazanymi na listingu 54, wgrać struktury bazy z pliku `createDB.sql` poleceniem `run`. Następnie załadować plik `initDB.sql` zawierający dane inicjujące za pomocą polecenia `run`. Pliki znajdują się w katalogu projektu `CAR/src/main/resources/`.

3.5.3 Nawiązanie połączenia z bazą danych

Kolejny krok to użycie polecenia `connect` w celu połączenia się z utworzoną wcześniej bazą danych, należy podać użytkownika i hasło, zgodnie z wartościami podanymi jako wartości właściwości `User` i `Password` na listingu 53.

```

[java@localhost ~]$ ij
wersja ij 10.13
ij> connect 'jdbc:derby://localhost:1527/CAR;create=true;traceFile=/home/java/trace.out';
ij> run '/CAR/src/main/resources/createDB.sql';
ij> run '/CAR/src/main/resources/initDB.sql';
ij> connect
'jdbc:derby://localhost:1527/CAR;user=cliniccar;password=cliniccar';
ij> exit;

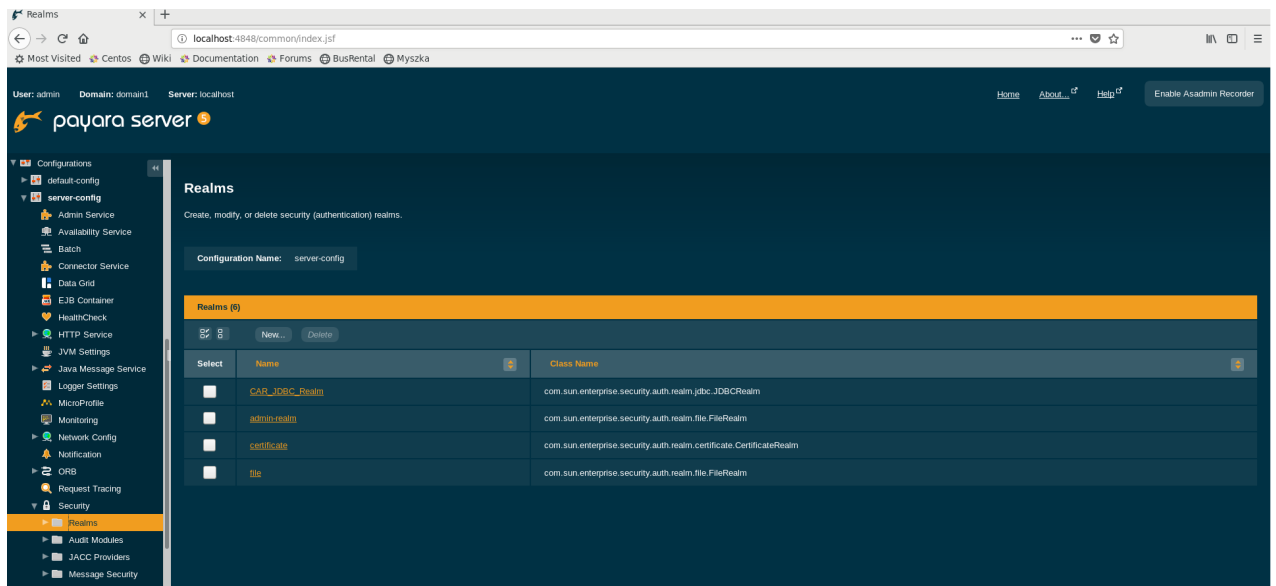
```

Listing 54: Tworzenie bazy danych JavaDB dla systemu operacyjnego Linux przy pomocy programu narzędziowego ij

3.5.4 Konfiguracja obszaru bezpieczeństwa na serwerze

Aby skonfigurować obszar bezpieczeństwa na serwerze, należy otworzyć przeglądarkę internetową i w pasek adresu podać adres URL lokalnego serwera <http://localhost:4848/common/index.jsf>. Po czym nacisnąć `Enter`.

Z menu po lewej stronie należy wybrać klikając kolejno: *Configurations* → *server-config* → *Security* → *Realms*. po czym wybrać przycisk *New* znajdujący się po prawej stronie, widoczny na rysunku 46.



Rysunek 46: Konfiguracja Security Realm na serwerze aplikacyjnym Payara

Otwarty formularz należy uzupełnić następującymi danymi i wycisnąć OK:

Realm Name: CAR_JDBC_Realm

Z listy **Class Name** wybrać: com.sun.enterprise.security.auth.realm.jdbc.JDBCRealm

JAAS Context: jdbcRealm

JNDI: jdbc/CAR_DS

User Table: AUTH_TABLE

User Name Column: LOGIN

Password Column: PASSWORD

Group Table: AUTH_TABLE

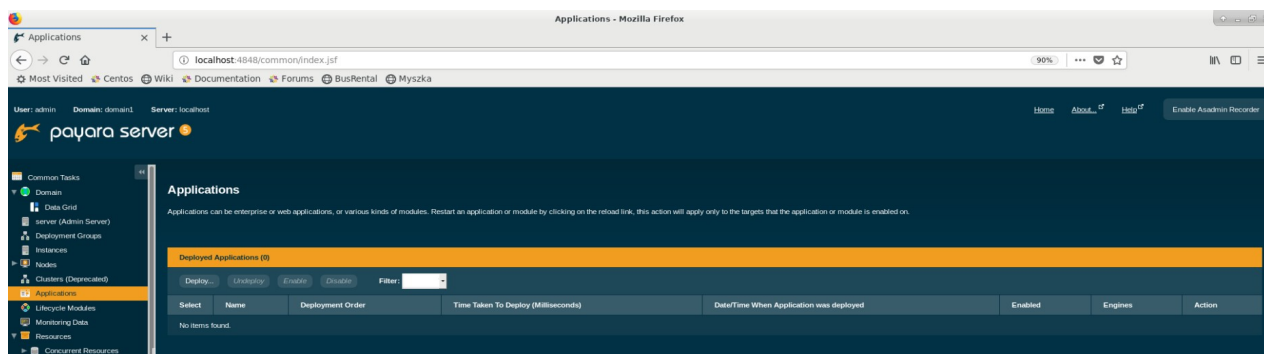
Group Name Column: LEVEL_OF_ACCESS

Digest Algorithm: SHA-256

Charset: UTF-8

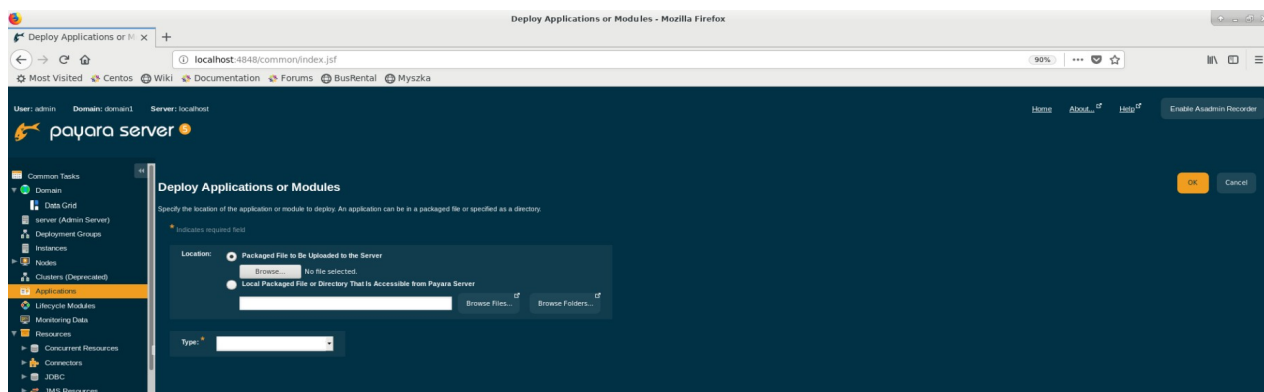
3.5.5 Wgranie aplikacji na serwer aplikacyjny i uruchomienie aplikacji

Należy z menu znajdującego się po lewej stronie ekranu wybrać przycisk *Application*, a następnie wybrać przycisk *Deploy*, co pokazuje rysunku 47.



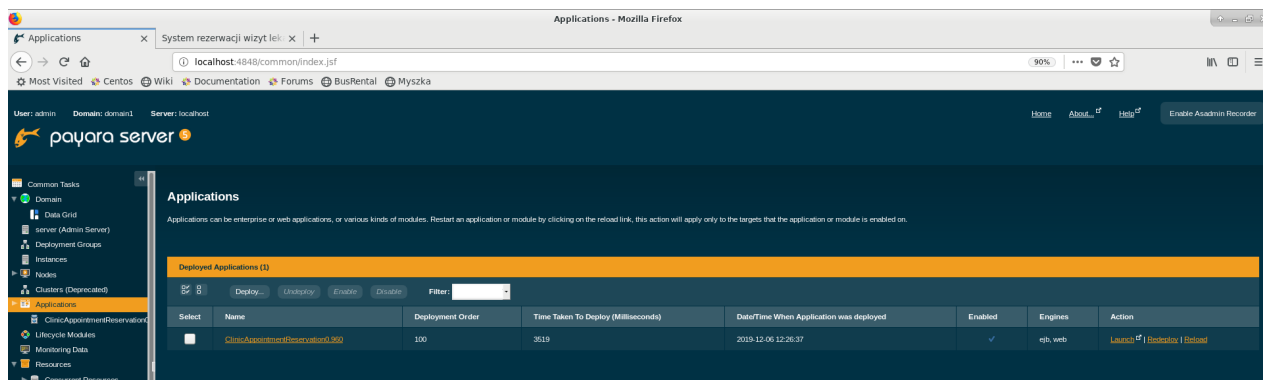
Rysunek 47: Wgrywanie aplikacji na serwer aplikacyjny

Poprzez użycie *Browse* wybrać plik *ClinicAppointmentReservation-1.0.war*, który należy pobrać z załączonego do raportu nośnika. Do pliku prowadzi ścieżka *ClinicAppointmentReservation/target/ClinicAppointmentReservation-1.0.war*. Po wybraniu pliku należy wybrać przycisk *Otwórz* w okienku na dole. Zatwierdzić wybór *OK*. Widok strony WWW serwera aplikacyjnego został przedstawiony w kroku drugim na rysunku 48



Rysunek 48: Wgranie aplikacji na serwer aplikacyjny

Aplikacja powinna być teraz widoczna na serwerze Payara. Aby ją uruchomić trzeba wybrać z kolumny *Action* przycisk *Launch*, co pokazuje rysunek 49.



Rysunek 49: Uruchomienie aplikacji na serwerze aplikacyjny

Jeśli cała procedura była przeprowadzona poprawnie, użytkownik aplikacji będzie miał możliwość zalogowania się na jedno z predefiniowanych kont, znajdujących się w pliku *initDB.sql*. Dane kont tzn. ich poziomy dostępu, loginy i hasła zostały ujawnione w tabeli 2

Tabela 2. Dane uwierzytelniania do utworzonego systemu.

Lp.	Poziom dostępu	Login	Hasło
1	Administrator	admin1	Qwerty1!
2	Recepcja	reception1	Qwerty1!
3	Pacjent	patient1	Qwerty1!

3.6 Podsumowanie

Celem pracy było zbudowanie systemu informatycznego umożliwiającego dokonywanie rezerwacji wizyt lekarskich w Klinice Medycyny Sportowej. Z funkcjonalności, jakie oferuje system mogą korzystać użytkownicy, którzy posiadają aktywne konta oraz, którym administrator przydzielił odpowiedni poziom dostępu tj. dokonał autoryzacji kont. Aplikacja posiada cztery poziomy dostępu. Podstawową funkcją systemu jest swobodne dodawanie rezerwacji wizyt, których dokonywać może Pacjent. Korzysta on przy tym z funkcjonalności kalendarza wykonanego przy użyciu technologii JavaScript. Może on także zarządzać własnymi rezerwacjami i w ograniczonym zakresie własnym kontem. Funkcjonalność zarządzania specjalizacjami dostępnymi w Klinice oraz specjalistami, do których mogą umawiać się pacjenci została wydzielona dla roli pracownika Recepcji. Użytkownik ten ma możliwość zarządzania rezerwacjami specjalistów oraz usuwać rezerwacje poszczególnych pacjentów. Zgodnie z założeniami, system przewiduje samodzielne rejestrowanie kont przez użytkowników, jednak konta stają się aktywne dopiero po autoryzowaniu ich przez Administratora. Przed autoryzacją administrator zobowiązany jest osobiście potwierdzić tożsamość nowo zarejestrowanego użytkownika. Nieuwierzytelniony użytkownik może zapoznać się z ofertą Kliniki.

System umożliwia dostęp do danych wielu uwierzytelnionym użytkownikom, zachowując przy tym spójność przetwarzania danych dzięki zastosowaniu przetwarzania transakcyjnego i użyciu mechanizmu blokad optymistycznych. Wielodostęp może być przyczyną licznych konfliktów. Poniżej zostały zaprezentowane przykłady błędów, które zostały wyeliminowane:

- możliwość edytowania i usuwania danych, które zostały w międzyczasie zmodyfikowane, dzięki zastosowanemu mechanizmowi blokad optymistycznych;
- możliwość modyfikacji poziomu dostępu, jeżeli wyświetlany na liście poziom dostępu jest już nieaktualny, dzięki zapamiętaniu, jakiego typu obiekt był wyświetlony i porównania go bezpośrednio przed wykonaniem metody z uaktualnionym typem znajdującym się w bazie;
- możliwość dezaktywacji wyłączonego wcześniej konta, dzięki zastosowaniu blokady optymistycznej;
- możliwość zarezerwowania terminu wizyty, która została już wcześniej zarezerwowana, dzięki obsłużeniu wyjątku dotyczącego naruszenia unikalności.

Aplikacja została wykonana w architekturze trójwarstwowej, dzięki której możliwe jest rozdzielenie warstwy widoku, dostępnego z poziomu przeglądarki internetowej, w której zastosowano szkielet JSF, warstwy logiki biznesowej, która została zaimplementowana za pomocą ziaren EJB oraz warstwy składowania danych w relacyjnej bazie danych, w której encje powstały z modelu wspieranego przez adnotacje mapowania obiektowo-relacyjnego JPA [2]. Do stworzenia aplikacji została użyta technologia Java Enterprise Edition, a osadzona została ona na serwerze aplikacyjnym Payara. Wszystkie dane są przechowywane w czterech tabelach zarządzanych przez system Apache Derby, do stworzenia których wykorzystano środowisko programistyczne NetBeans.

Podsumowując, wszystkie założenia projektu zostały osiągnięte. Stworzony system zautomatyzuje proces rezerwacji wizyt u lekarzy specjalistów, który odbywał się dotychczas za pośrednictwem telefonu lub osobistego stawiennictwa. Aplikacja jest także doskonałą bazą do dalszej rozbudowy. Zagadnienia, które warto rozważyć w przypadku dalszego rozwoju oprogramowania zostały ujęte w poniższych podpunktach:

- stworzenie interaktywnego grafiku pracy specjalistów;
- stworzenie poziomu dostępu dla Specjalistów, którzy będą mogli sami zarządzać swoim grafikiem, dokonywać rezerwacji dla pacjentów jeszcze podczas ich wizyty w gabinecie;
- system mailingowy i smsowy, m. in. w celu wysyłania przypomnień o nadchodzących wizytach;
- wybór typu rezerwowanej wizyty i jej długości: np. konsultacja lub zabieg, odpowiednio 20 lub 40 minut;
- filtrowanie i sortowanie list rezerwacji.

4 Źródła

Bibliografia

1. Basham Bryan, Sierra Kathy, Bates Bart: Head First. Servlets & JSP.. Edycja polska., Wydanie I, Helion, 2005
2. Baue Ch., King G., Gregory G.: Java Persistence. Programowanie aplikacji bazodanowych w Hibernate. Wydanie II, Wydanie , Helion, 2016
3. Beighley Lynn: Head First. SQL. Edycja polska., Wydanie I, Helion, 2011
4. Burke Bill, Monson-Haefel Richard: Enterprise JavaBeans 3.0, Wydanie V, Helion, 2007
5. Downey Allen B., Mayfield Chris: Myśl w języku Java! Nauka programowania., Wydanie I, Helion, 2017
6. Fowler Martin: Patterns of Enterprise Application Architecture. Data Transfer Object, Wydanie I, Addison-Wesley, 2010
7. Fowler Martin: Patterns of Enterprise Applications, Wydanie , Addison-Wesley, 2010
8. Heffelfinger, David R. : Java EE 6. Tworzenie aplikacji w NetBeans 7, Wydanie , Helion, 2014
9. Horstmann Cay S.: Java. Techniki zaawansowane, Wydanie X, Helion, 2017
10. Horstmann Cay S.: Java. Podstawy., Wydanie X, Helion, 2016
11. Kuo Peter, Pence John: Sieci komputerowe, Wydanie I, Helion, 2000
12. Lis Marcin: Praktyczny kurs Java, Wydanie IV, Helion, 2015
13. McLaughlin Brett D., Pollice Gary, West David: Head First. Object-Oriented Analysis & Design.. Edycja polska., Wydanie I, Helion, 2008
14. Rychlicki-Kicior Krzysztof: Java EE. Programowanie aplikacji www., Wydanie II, Helion, 2015
15. Sierra Kathy, Bates Bert: Head First. Java. Edycja polska., Wydanie II, Helion, 2011
- 16: , Bootstrap 3 Datepicker v4 Docs, , <https://eonasdan.github.io/bootstrap-datetimepicker/>

5 Spis załączników

Tabele

Tabela 1. Tabela krzyżowa poziomów dostępu i przypadków użycia.....	9
Tabela 2. Dane uwierzytelniania do utworzonego systemu.....	62

Wykaz rysunków

Rysunek 1: Diagram przypadków użycia dla poziomu dostępu: Gość.....	7
Rysunek 2: Diagram przypadków użycia dla poziomu dostępu: Administrator.....	7
Rysunek 3: Diagram przypadków użycia dla poziomu dostępu: Recepcja.....	8
Rysunek 4: Diagram przypadków użycia dla poziomu dostępu: Pacjent.....	8
Rysunek 5: Zarejestruj konto.....	10
Rysunek 6: Zaloguj się.....	11
Rysunek 7: Resetowanie hasła (Wpisz login).....	11
Rysunek 8: Resetowanie hasła (Wpisz odpowiedź).....	11
Rysunek 9: Resetowanie hasła (Wpisz hasło).....	11
Rysunek 10: Wyloguj się / Wyświetl swój login.....	11
Rysunek 11: Wyświetl dane własnego konta.....	12
Rysunek 12: Edytuj dane swojego konta.....	12
Rysunek 13: Zmiana własnego hasła.....	13
Rysunek 14: Wyświetl listę nieautoryzowanych kont.....	13
Rysunek 15: Wyświetl listę autoryzowanych kont.....	14
Rysunek 16: Zmień dane dowolnego konta.....	14
Rysunek 17: Zmień hasło dowolnego konta.....	15
Rysunek 18: Edytuj dane specjalizacji.....	15
Rysunek 19: Dodaj specjalizację.....	15
Rysunek 20: Wyświetl listę specjalizacji.....	15
Rysunek 21: Dodaj specjalistę.....	16
Rysunek 22: Wyświetl listę specjalistów.....	16
Rysunek 23: Edytuj dane specjalisty.....	17
Rysunek 24: Wyświetl listę rezerwacji specjalisty.....	17
Rysunek 25: Wyświetl listę wszystkich aktualnych rezerwacji.....	17
Rysunek 26: Wyświetl listę wszystkich przeszłych rezerwacji.....	18
Rysunek 27: Utwórz rezerwację – wybór specjalizacji.....	18
Rysunek 28: Utwórz rezerwację – wybór specjalisty.....	18
Rysunek 29: Utwórz rezerwację – wybór daty.....	18
Rysunek 30: Utwórz rezerwację – wybór godziny.....	19
Rysunek 31: Edytuj rezerwację.....	19
Rysunek 32: Wyświetl listę swoich aktualnych rezerwacji.....	19
Rysunek 33: Wyświetl listę swoich przeszłych rezerwacji.....	20
Rysunek 34: Wyskakujące okno do potwierdzania akcji usuwania zamówień.....	20
Rysunek 35: Diagram komponentów dla przypadków użycia CRUD związanych z kontami użytkowników.....	21
Rysunek 36: Relacyjny model bazy danych.....	34
Rysunek 37: Diagram klas encyjných.....	37
Rysunek 38: Wygląd sekcji nagłówka strony.....	50

Rysunek 39: Wygląd paska menu dla użytkownika nieuwierzytelnionego.....	50
Rysunek 40: Wygląd paska menu dla poziomu dostępu: Administrator.....	50
Rysunek 41: Wygląd paska menu dla poziomu dostępu: Recepcja.....	50
Rysunek 42: Wygląd paska menu dla poziomu dostępu: Pacjent.....	50
Rysunek 43: Wygląd sekcji zawartości głównej na przykładzie strony mainPage.xhtml i sekcji stopki.....	50
Rysunek 44: Fragment pliku messages_pl.properties.....	52
Rysunek 45: Strona newReservation.xhtml - wybór zakresu dat z kalendarza.....	57
Rysunek 46: Konfiguracja Security Realm na serwerze aplikacyjnym Payara.....	60
Rysunek 47: Wgrywanie aplikacji na serwer aplikacyjny.....	61
Rysunek 48: Wgranie aplikacji na serwer aplikacyjny.....	61
Rysunek 49: Uruchomienie aplikacji na serwerze aplikacyjny.....	62

Wykaz listingów

Listing 1: Fragment kodu strony registerAccount.xhtml - formularz odpowiedzialny za tworzenie konta użytkownika.....	22
Listing 2: Fragment kodu klasy AccountRegistrationPageBean.....	23
Listing 3: Fragment kodu klasy AccountControllerBean.....	23
Listing 4: Fragment kodu klasy AccountEndpoint odpowiedzialny za przeniesienie danych z obiektu DTO na obiekt Encji i wywołanie metody „create” w klasie AccountFacade.....	24
Listing 5: Fragment kodu klasy AccountFacade – metoda create.....	24
Listing 6: Fragment kodu klasy AbstractFacade.....	25
Listing 7: Fragment kodu strony listNewAccounts.xhtml odpowiedzialny za wyświetlenie listy wszystkich nieautoryzowanych kont.....	25
Listing 8: Fragment kodu klasy ListNewAccountsPageBean odpowiedzialny za przetworzenie i wyświetlenie listy wszystkich nieautoryzowanych kont.....	26
Listing 9: Fragment kodu klasy AccountControllerBean – metoda listNewAccounts().....	27
Listing 10: Fragment kodu klasy AccountEndpoint.....	27
Listing 11: Metoda klasy AccountFacade.....	27
Listing 12: Kwerenda klasy encyjnej Account.....	28
Listing 13: Fragment kodu strony listAuthorizedAccount.xhtml odpowiedzialny za wyświetlenie przycisków umożliwiających aktywację i dezaktywację konta autoryzowanego użytkownika....	28
Listing 14: Fragment kodu klasy ListAuthorizedAccountPageBean odpowiedzialny za wywołanie metod aktywacji i dezaktywacji konta.....	29
Listing 15: Metoda kontrolera AccountControllerBean odpowiedzialna za sprawdzenie czy metoda aktywacji konta nie została wykonana ponownie.....	29
Listing 16: Metoda klasy AccountEndpoint odpowiedzialna za wywołanie metody wyszukiwanej findByLogin w klasie AccountFacade i ustawieniu stanu aktywności konta.....	30
Listing 17: Metoda edit klasy AccountFacade.....	30
Listing 18: Fragment kodu strony listNewAccounts.xhtml odpowiedzialny za wyświetlenie przycisku do kasowania konta.....	31
Listing 19: Fragment klasy ListNewAccountsPageBean.....	31
Listing 20: Metoda klasy AccountControllerBean odpowiedzialna na sprawdzenie czy nie została podjęta próba ponownego wykasowania tego samego obiektu.....	32
Listing 21: Metoda klasy AccountEndpoint odpowiedzialna na wywołanie kwerendy wyszukiwanej obiekt z bazy danych oraz wywołująca polecenie remove mające na celu usunięcie tego obiektu.....	32
} Listing 22: Metoda klasy AccountFacade odpowiedzialna za obsługę wyjątków i usunięcie odpowiedniej krotki z bazy danych.....	33

Listing 23: Metoda klasy AccountFacade odpowiedzialna za usunięcie z bazy danych wybranej krotki.....	33
Listing 24: Plik deskryptora składowania persistence.xml.....	35
Listing 25: Klasa AbstractEntity - Implementacja bazowej klasy dla encji.....	36
Listing 26: Fragment klasy encyjnej Account.....	37
Listing 27: Fragment klasy abstrakcyjnej AbstractFacade.....	38
Listing 28: Fragment klasy SpecializationFacade.....	39
Listing 29: Klasa abstrakcyjna AbstractEndpoint.....	41
Listing 30: Fragment klasy SpecialistEndpoint.....	42
Listing 31: Fragment klasy LoggingInterceptor – mechanizm przechwytyjący.....	43
Listing 32: Fragment pliku konfiguracyjnego web.xml - deskryptora wdrożenia aplikacji.....	44
Listing 33: Fragment klasy wyjątków AppBaseException.....	45
Listing 34: Fragment klasy ReservationEndpoint metoda addReservation().....	47
Listing 35: Fragment klasy SpecialistDTO.....	48
Listing 36: Fragment szablonu mainTemplate.xhtml.....	49
Listing 37: Fragment strony mainPage.xhtml.....	49
Listing 38: Fragment pliku deskryptora wdrożenia web.xml odpowiedzialny za konfigurację zasobu dla internacjonalizacji.....	51
Listing 39: Fragment pliku faces_config.xml.....	51
Listing 40: Fragmenty klasy ContextUtils.....	52
Listing 41: Fragment pliku login.xhtml.....	52
Listing 42: Fragment deskryptora wdrożenia web.xml.....	53
Listing 43: Fragment pliku mainTemplate.xhtml przedstawiający podział w widoku menu.....	53
Listing 44: Fragmenty kodu deskryptora wdrożenia web.xml.....	54
Listing 45: Fragmenty deskryptora wdrożenia aplikacji glassfish-web.xml.....	54
Listing 46: Fragment klasy encyjnej Account przedstawiający metodę setPassword().....	54
Listing 47: Fragment pliku addSpecialist.xhtml – walidacja wprowadzanych danych.....	55
Listing 48: Fragment klasy NewAccountPageBean.....	55
Listing 49: Fragment strony registerAccount.xhtml.....	56
Listing 50: Fragment szablonu mainTemplate.xhtml odpowiedzialny za wyświetlenie paska menu.....	56
Listing 51: Fragment strony newReservation3.xhtml.....	57
Listing 52: Fragment klasy MainApplicationPageBean.....	58
Listing 53: Fragment pliku konfiguracyjnego glassfish-resource.xml.....	59
Listing 54: Tworzenie bazy danych JavaDB dla systemu operacyjnego Linux przy pomocy programu narzędziowego ij.....	59