Politechnika Łódzka
Instytut Informatyki

RAPORT Z PROJEKTU KOŃCOWEGO

STUDIÓW PODYPLOMOWYCH

NOWOCZESNE APLIKACJE BIZNESOWE JAVA EE

SYSTEM WYPOŻYCZANIA BUSÓW W FIRMIE PRZEWOZOWEJ

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej
Opiekun: dr inż. Mateusz Smoliński
Słuchacz: Piotr Graczykowski

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

Celem pracy jest utworzenie wielodostępnego systemu informatycznego, umożliwiającego obsługę zamówień wypożyczania busów w firmie przewozowej. System ma pozwalać zarejestrowanym klientom, zgłaszać rezerwację busa na dany okres. Planista pracujący w firmie, ma być odpowiedzialny za weryfikację powyższych zgłoszeń, a także za zarządzanie busami floty.

System informatyczny zostanie utworzony z wykorzystaniem technologii Java Enterprise Edition i będzie złożony z aplikacji internetowej oraz relacyjnej bazy danych. Aplikacja internetowa zostanie osadzona na serwerze aplikacyjnym Payara i będzie dostępna poprzez interfejs użytkownika z poziomu współczesnej przeglądarki internetowej. Dane aplikacji będą przechowywane w relacyjnej bazie danych zlokalizowanej w systemie zarządzania bazami danych Java DB.

Zakres projektu obejmuje:

- Opracowanie założeń i wymagań funkcjonalnych tworzonego systemu,
- Wybór odpowiedniego stosu technologicznego,
- Przygotowanie struktur relacyjnej bazy danych,
- Zaprojektowanie i implementacja aplikacji internetowej,
- Przygotowanie danych inicjujących dla relacyjnej bazy danych,
- Uruchomienie systemu informatycznego w środowisku serwera aplikacyjnego.

2 Założenia projektu

Rozdział zawiera prezentacje wszystkich założeń prezentowanego projektu, technologie i narzędzia zastosowane przy jego realizacji , jego budowę, wymagania i prezentację przykładowego przepływu informacji.

2.1 Wersje zastosowanych technologii i narzędzi

System informatyczny zostanie zbudowany z wykorzystaniem następujących narzędzi i technologii środowiska produkcyjnego i developerskiego:

- Serwer aplikacyjny Payara 5.184
- Java DB (Derby) 10.13
- NetBeans IDE 8.2
- Język programowania Java 8 [11]
- JDK 1.8_201
- Java Enterprise Edition 7.0
- JavaScript 1.8
- jQuery 3.2.1
- Maven 3.6.0
- JSF 2.2
- EJB 3.2
- Java Persistence API 2.1
- Modelio 3.7
- HTML 5 i CSS 3
- Bootstrap 3.3.7

2.2 Wymagania funkcjonalne

Realizowany projekt będzie posiadał 4 poziomy dostępu. W jego założeniach przyjęto, że każdy użytkownik systemu musi dysponować indywidualnym, autoryzowanym przez administratora kontem, które może mieć przypisany co najwyżej jeden poziom dostępu. Każdemu z poziomów zostały przypisane inne zbiory przypadków użycia. Użytkownik nieuwierzytelniony posiada poziom dostępu **Gość**, ma możliwość utworzenia i zalogowania się na własne konto. Może również wyświetlić ofertę firmy i 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 zamówieniami, busami i innymi kontami.

- **Planista** – zarządza flotą busów, może je dodawać, usuwać, aktywować, dezaktywować i edytować ich dane. Zajmuje się też obsługą zamówień klientów na wypożyczenia busów min. ich odwoływaniem i masowym usuwaniem. Może wyświetlić zamówienia w kilku dostępnych konfiguracjach. Ma też dostęp do listy klientów i przypisanych do nich zamówień.

- **Klient** – ma możliwość utworzenia zgłoszenia na wypożyczenie autobusu poprzez wybór busa i okresu zamówienia poprzez interaktywny kalendarz. Ma dostęp do listy swoich aktualnych i przeszłych zamówień. Może edytować lub usuwać swoje aktualne zamówienia.

Niezależnie od wybranego do przypisania poziomu 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. W przypadku autoryzowania konta z poziomem dostępu od Klienta wymagane jest również, okazanie prawa jazdy z odpowiednimi uprawnieniami do kierowania busami.

System będzie posiadał też następujące funkcjonalności:

1. Rezerwacja busów obejmowała będzie wybrany zakres dni w systemie dobowym.
2. Maksymalny zakres zamówienia to dwa tygodnie.
3. Klient może posiadać tylko jedno zamówienie w danym czasie.
4. Bus może być przypisany tylko do jednego zamówienia w danym czasie.
5. Nie można składać zamówień na dni przeszłe.
6. Początek ani koniec zamówienia nie mogą przypadać na dni ustawowo wolne od pracy (ale mogą być one zawarte w okresie zamówienia).
7. Koniec okresu zamówienia może przypadać maksymalnie na 6 miesięcy od obecnego dnia.
8. System nie pozwoli złożyć zamówienia na dzień obecny, gdy minie godzina 20.
9. Zamówione busy należy odbierać i zwracać w w godzinach pracy firmy od 8 do 20.
10. System nie pozwoli zamówić klientowi dezaktywowanego busa do momentu jego ponownej aktywacji.
11. Każdy uwierzytelniony użytkownik ma dostęp do informacji o swoim loginie i przypisanym poziomie dostępu.
12. Każdy z uwierzytelnionych użytkowników może wyświetlić, edytować dane, oraz zmieniać hasło własnego konta. Do przeprowadzenia zmian konieczne jest podanie swojego obecnego hasła.
13. Dezaktywowanie użytkownika, powoduje natychmiastową utratę dostępu do krytycznych operacji na jego koncie, nawet gdy jest on obecnie zalogowany. Po zakończeniu sesji nie ma możliwości ponownego zalogowania się na własne konto.

2.3 Wymagania niefunkcjonalne

System będzie spełniał następujące wymaganie niefunkcjonalne:

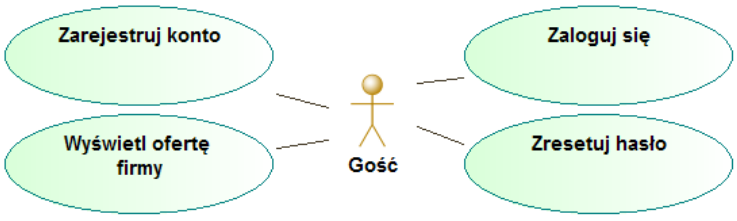
1. Aplikacja będzie zbudowana w architekturze trójwarstwowej, z podziałem na warstwę składowania danych, warstwę logiki biznesowej i warstwę widoku.
2. System będzie wymagał uwierzytelnienia poprzez podanie loginu i hasła.
3. Wzorce do uwierzytelniania będą przechowywane w relacyjnej bazie danych, przy czym hasła będą przechowywane jako skrót SHA-256.
4. Uwierzytelniony użytkownik będzie miał możliwość wylogowania się.
5. Dane aplikacji będą przechowywane w relacyjnej bazie danych.
6. Do odwzorowania modelu relacyjnego w modelu obiektowym zostanie wykorzystany standard mapowania obiektowo-relacyjnego JPA 2.1.
7. System będzie zapewniał wielodostęp.
8. Spójność danych w systemie zapewnią transakcje i blokady optymistyczne.
9. System zapewni mechanizm zgłaszania komunikatów o występujących zdarzeniach i gromadzenia ich w dzienniku zdarzeń.
10. Interfejs użytkownika będzie zapewniony dzięki dynamicznie generowanym stronom WWW dostępnym poprzez współczesną przeglądarkę internetową.
11. System będzie posiadał internacjonalizację ograniczoną do interfejsu użytkowania.
12. System będzie posiadał polską i angielską wersję językową.
13. System nie będzie obsługiwał znaków diakrytycznych.
14. System operacyjny, w którym będzie działał serwer aplikacyjny, będzie posiadał aktualny czas.
15. Aplikacja nie będzie zgodna z wymogami określonymi w Rozporządzeniu o Ochronie Danych Osobowych.

2.4 Przypadki użycia dla różnych poziomów dostępu

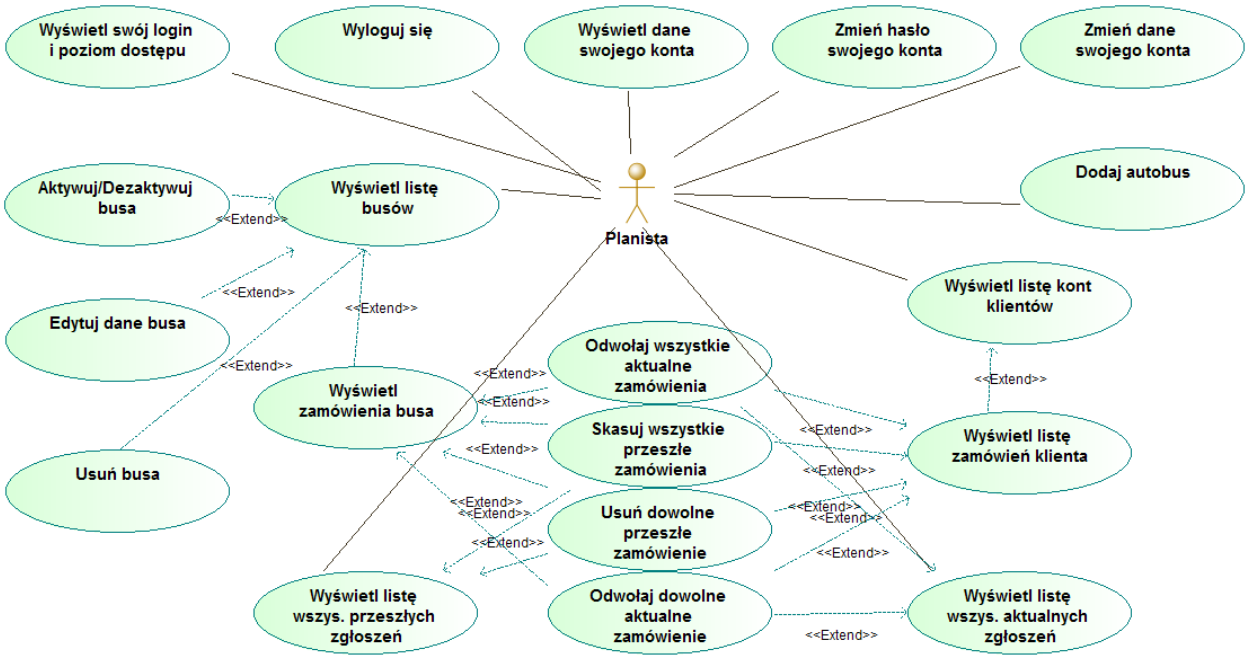
Poniższy rozdział przedstawia szczegółowe informacje na temat występujących w aplikacji przypadkach użycia w postaci diagramów i tabel.

2.4.1 Diagramy przypadków użycia

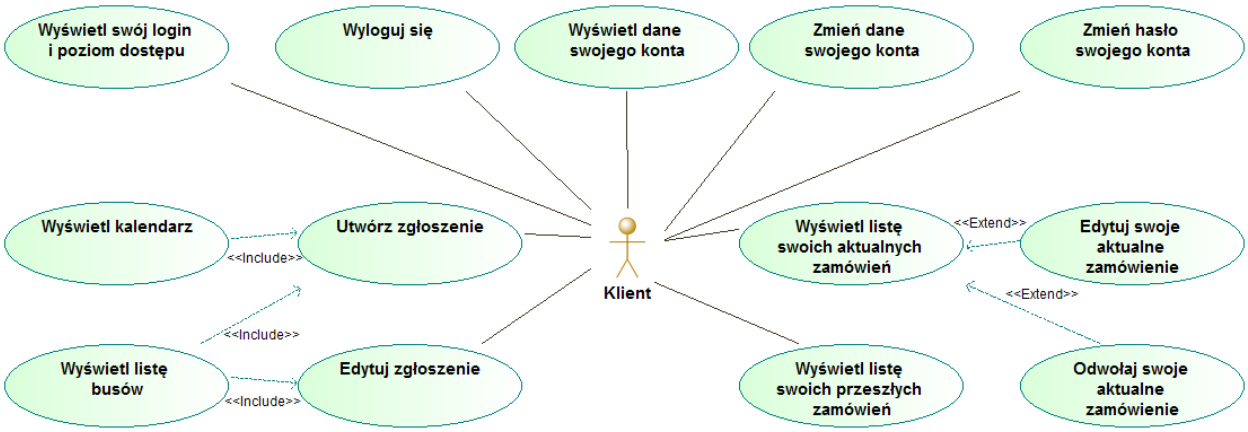
Opis funkcjonalności oferowanej przez system informatyczny przedstawiono w formie diagramów przypadków użycia dla poszczególnych poziomów dostępu[15]: Gość (rys.1), Planista (rys. 2), Klient (rys. 3), Administrator (rys. 4).



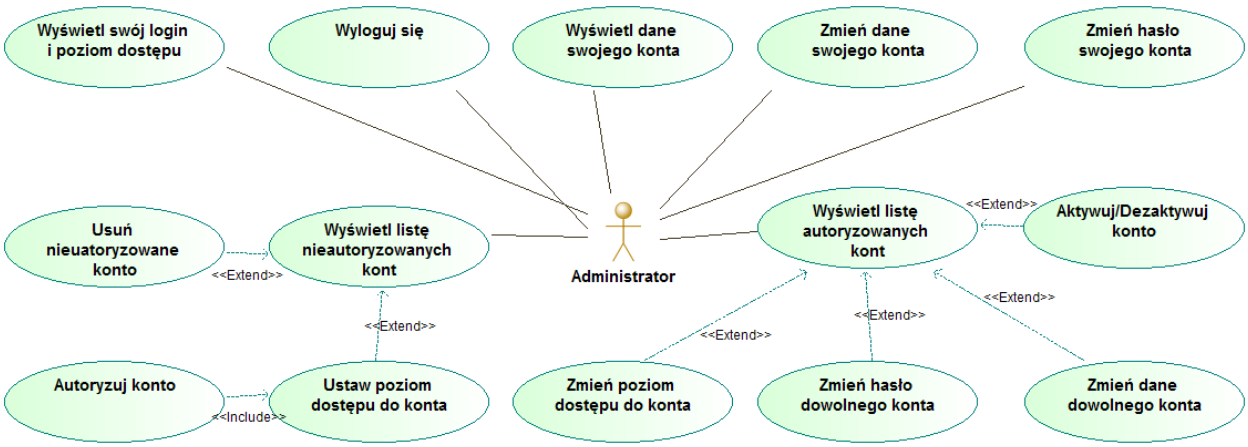
Rysunek 1: Diagram przypadków użycia dla poziomu dostępu: Gość



Rysunek 2: Diagram przypadków użycia dla poziomu dostępu: Planista



Rysunek 3: Diagram przypadków użycia dla poziomu dostępu: Klient



Rysunek 4: Diagram przypadków użycia dla poziomu dostępu: Administrator

2.4.2 Tabela krzyżowa ról i przypadków użycia

W tabeli 1 przedstawiono zestawienie wszystkich przypadków użycia razem z odpowiadającymi im poziomami dostępu.

Tabela 1. Tabela krzyżowa poziomów dostępu i przypadków użycia.

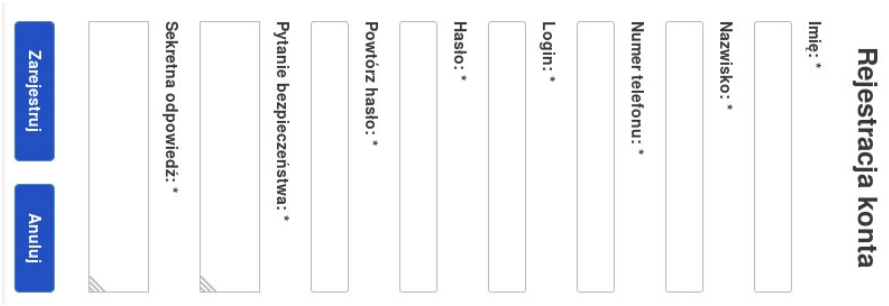
Lp.	Przypadek użycia	Gość	Admin	Planista	Klient
1	Zarejestruj konto	x			
2	Zaloguj się	x			
3	Zresetuj hasło	x			
4	Wyświetl ofertę firmy	x			
5	Wyloguj się		x	x	x
6	Wyświetl dane swojego konta		x	x	x
7	Zmień dane swojego konta		x	x	x
8	Zmień hasło swojego konta		x	x	x
9	Wyświetl swój login i poziom dostępu		x	x	x
10	Wyświetl listę nieautoryzowanych kont		x		
11	Usuń nieautoryzowane konto		x		
12	Ustaw poziom dostępu i autoryzuj konto		x		
13	Wyświetl listę autoryzowanych kont		x		
14	Zmień poziom dostępu dla konta		x		
15	Zmień dane dowolnego konta		x		
16	Aktywuj/Dezaktywuj konto		x		
17	Zmień hasło dowolnego konta		x		
18	Dodaj busa			x	
19	Wyświetl listę busów			x	
20	Edytuj dane busa			x	
21	Aktywuj/Dezaktywuj busa			x	
22	Usuń busa			x	
23	Wyświetl listę zamówień busa			x	
24	Wyświetl listę klientów			x	
25	Wyświetl listę zamówień klienta			x	
26	Wyświetl listę wszystkich aktualnych zamówienie			x	
27	Wyświetl listę wszystkich przeszłych zamówienie			x	
28	Utwórz zamówienie				x
29	Edytuj zamówienie				x
30	Wyświetl listę swoich aktualnych zamówień				x
31	Wyświetl listę swoich przeszłych zamówień				x

Lp.	Przypadek użycia	Gość	Admin	Planista	Klient
32	Odwołaj swoje aktualne zamówienie				x
33	Odwołaj dowolne aktualne zamówienie			x	
34	Usuń dowolne przeszłe zamówienie			x	
35	Skasuj wszystkie aktualne zamówienia			x	
36	Odwołaj wszystkie przeszłe zamówienia			x	

2.4.3 Opis przypadków użycia

Poniżej przedstawiono szczegółowy opis wszystkich przypadków użycia, wraz z rysunkami prezentującymi poszczególny przypadek w widoku przeglądarki [13].

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



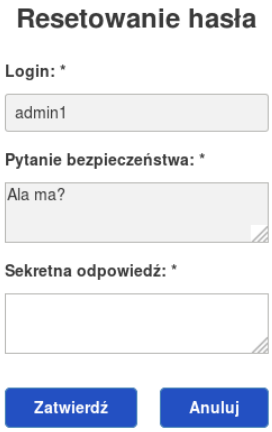
Rysunek 5: Zarejestruj konto

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.

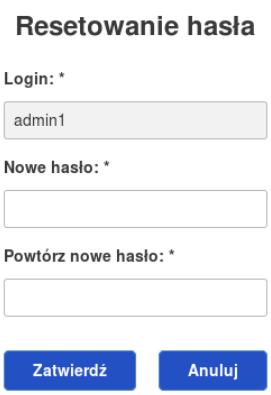
Rysunek 6: Zaloguj się

Zresetuj hasło - jest to funkcja dostępna dla użytkownika, chcącego ustawić nowe hasło. Składa się z trzech odrębnych stron. Pierwszej strona widoczna na rysunku 7 użytkownik podaje swój login, przypisany do konta i potwierdza przyciskiem „Zatwierdź". Następnie, po sprawdzeniu w bazie danych czy podany login istnieje, następuje przekierowanie na kolejną stronę pokazaną na rysunku 8, gdzie wyświetlone zostaje pytanie do resetowania hasła. Należy podać odpowiedź na pytanie, które zostało podane przez użytkownika podczas rejestracji konta i nacisnąć przycisk „Zatwierdź". Następuje kolejne przekierowanie na stronę, na której należy wpisać nowe hasło oraz powtórzyć je, celem sprawdzenia czy nie doszło do pomyłki, co przedstawia rysunek 9. Jeżeli wszystkie dane zostały poprawnie wpisane, nowe hasło użytkownika zostaje zapisane w bazie danych w postaci niejawnej [13].

Rysunek 7: Resetowanie hasła (Podanie loginu)

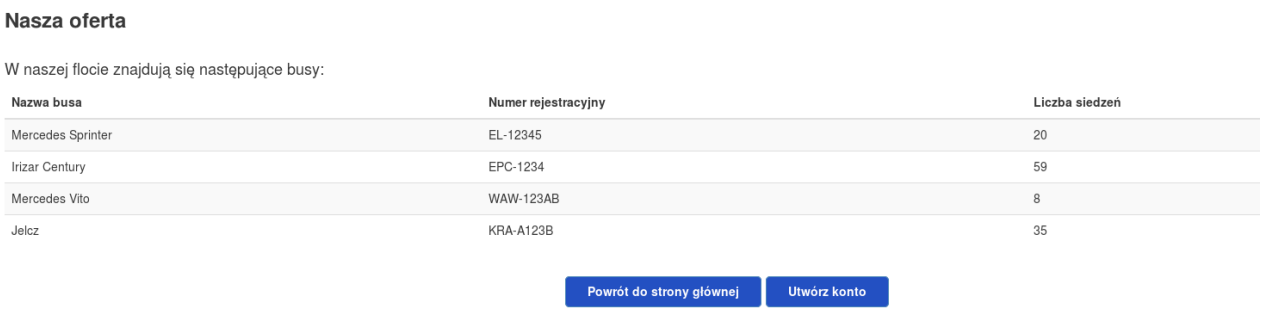


Rysunek 8: Resetowanie hasła (Pytanie kontrolne)



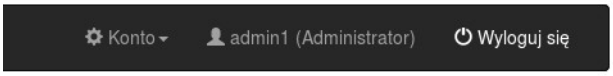
Rysunek 9: Resetowanie hasła (Podanie hasła)

Wyświetl ofertę firmy – nieuwierzytelniony użytkownik ma możliwość zapoznania się z ofertą firmy przedstawioną na rysunku 10. Składa się ona z listy wszystkich aktualnie dostępnych busów, oraz ich numerów rejestracyjnych wraz z informacją o liczbie siedzeń w każdym z nich.



Rysunek 10: Wyświetl ofertę firmy

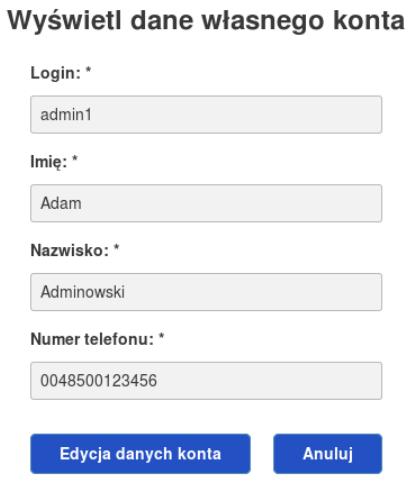
Wyloguj się - użytkownik po kliknięciu przycisku „Wyloguj się", znajdującym się na pasku menu widocznym na rysunku 11, traci dostęp do funkcjonalności przypisanych dla przydzielonego mu poziomu dostępu,. Tym samym, zyskuje dostęp do funkcjonalności przypisanych użytkownikowi nieuwierzytelnionemu.



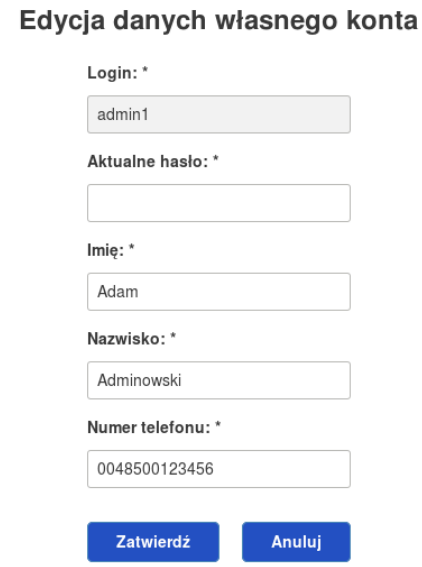
Rysunek 11: Wyloguj się

Wyświetl dane swojego konta – formularz widoczny na rysunku 12 pozwala użytkownikowi na wgląd w informacje o własnym koncie. Możliwe jest też przejście do edycji danych po wybraniu przycisku „Edycja danych konta”.

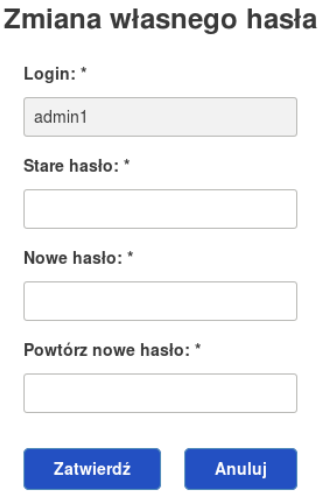
Zmień dane swojego konta - funkcjonalność ta daje użytkownikowi możliwość edycji swoich danych osobowych i numeru telefonu widocznych na rysunku 13. Aby to zrobić, konieczne jest podanie aktualnego hasła w celu weryfikacji użytkownika.



Rysunek 12: Wyświetl dane własnego konta



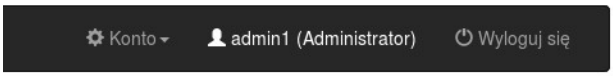
Rysunek 13: Edycja danych własnego konta



Rysunek 14: Zmiana własnego hasła

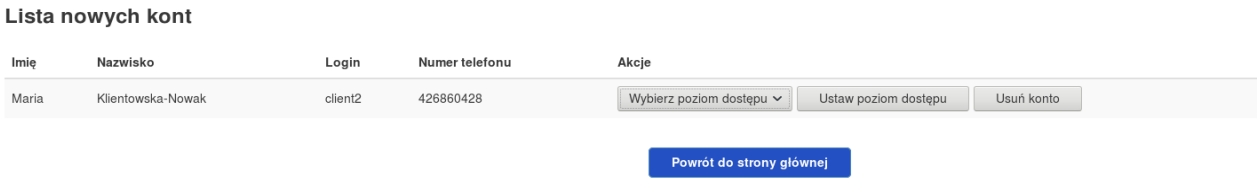
Zmień hasło swojego konta - przypadek ten dostępny jest dla uwierzytelnionego użytkownika. Po poprawnym podaniu przez niego starego hasła i dwukrotnym wprowadzeniu nowego, co przedstawia rysunek 14, zapisywane jest ono w bazie danych w formie niejawnej. Przed zapisaniem zmiany wpisane stare hasło jest porównywane z tym, które obecnie znajduje się w bazie.

Wyświetl swój login i poziom dostępu – funkcja dla uwierzytelnionych użytkowników, która powoduje samoczynne wywołanie metody wyświetlającej na pasku menu, ukazanym na rysunku 15, login oraz przypisany do konta poziom dostępu.



Rysunek 15: Wyświetl swój login i poziom dostępu

Wyświetl listę nieautoryzowanych kont - umożliwia administratorowi wgląd w nowo zarejestrowane, czekające na autoryzację konta, a także wykonanie związanych z nimi akcji, co przedstawia rysunek 16.

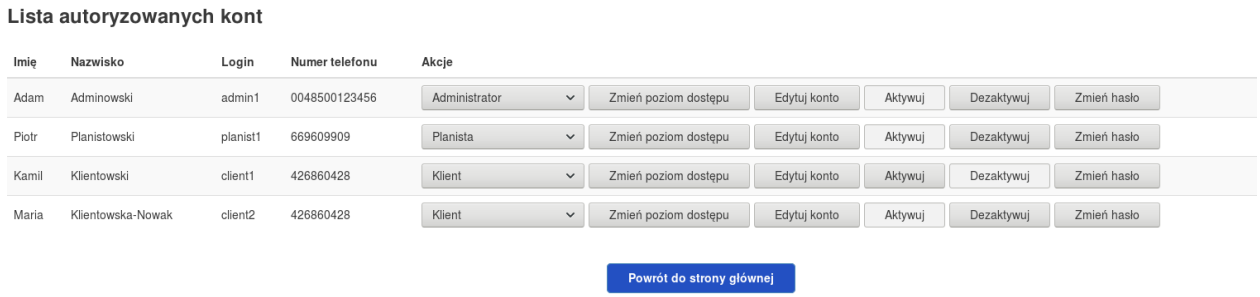


Rysunek 16: Wyświetl listę nieautoryzowanych kont

Usuń nieautoryzowane konto - powoduje usunięcie wybranego z listy konta z bazy danych poprzez wybranie przycisku „Usuń konto” widocznego na rysunku 16. Usunięcie jest możliwe wyłącznie dla kont jeszcze nieautoryzowanych przez administratora.

Ustaw poziom dostępu i autoryzuj konto - odbywa się przez wybranie przez administratora poziomu dostępu dla konta użytkownika, który został wybrany z listy. Z rozwijanej listy należy wybrać odpowiedni poziom dostępu, a następnie kliknąć „Ustaw poziom dostępu”, jak pokazano na rysunku 16. Po przydzieleniu poziomu dostępu konto nie jest już widoczne na liście nowo zarejestrowanych kont i trafia na listę kont użytkowników autoryzowanych, widoczną na rysunku 17.

Wyświetl listę autoryzowanych kont - powoduje pobranie z bazy i wyświetlenie listy autoryzowanych kont użytkowników, która jest widoczna na rysunku 17. Podobnie jak w przypadku listy nowo zarejestrowanych kont, można wybrać konto i dokonać zmian.



Rysunek 17: Wyświetl listę autoryzowanych kont

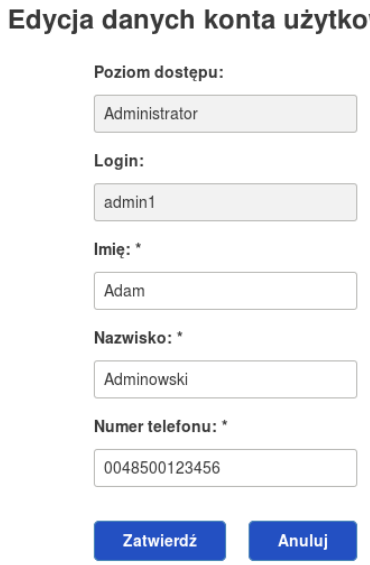
Zmień poziom dostępu dla konta – pozwala zmienić poziom dostępu dla wybranego z listy konta użytkownika, poprzez kliknięcie przycisku „Zmień poziom dostępu” widocznego na rysunku 17. Zmiana ta powoduje przepisanie wszystkich danych obecnego konta do nowego konta, ustawienie w nim wybranego poziomu dostępu i dodanie go do bazy danych. Obecne konto zostaje usunięte z bazy danych. Operacja ta jest możliwa wyłącznie wtedy, kiedy konto użytkownika nie posiada żadnych istniejących relacjami w tabelach bazy danych.

Aktywuj/Dezaktywuj konto - dezaktywacja ma zastosowanie w przypadku gdy z jakiegoś powodu administrator chce zablokować użytkownikowi dostęp do aplikacji, powoduje natychmiastową utratę dostępu do krytycznych operacji na jego koncie, nawet gdy użytkownik jest obecnie zalogowany. Po zakończeniu sesji nie ma możliwości ponownego zalogowania się na dezaktywowane konto, aż do jego ponownej aktywacji. Na rysunku 17 przedstawiono widok listy autoryzowanych kont z jednym dezaktywowanym kontem. Przyciski „Aktywuj” i „Dezaktywuj” działają naprzemiennie, po wciśnięciu dany przycisk zostaje zablokowany, aby nie było możliwe jego dwukrotne wciśnięcie.

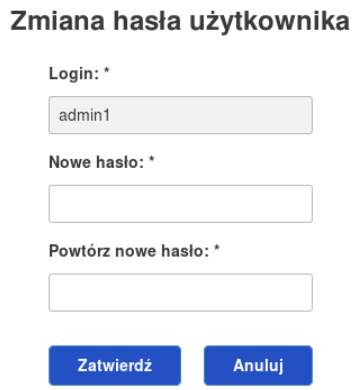
Zmień dane dowolnego konta – funkcja stosowana, gdy niezbędne jest wprowadzenie zmian w danych osobowych użytkownika przez administratora. Formularz, przestawiony na rysunku 18, daje możliwość zmiany imienia, nazwiska i numeru telefonu. Login jest

unikalny i nie może zostać zmieniony, dlatego pole zablokowane jest do edycji, o czym świadczy jego szara barwa.

Zmień hasło dowolnego konta - pozwala administratorowi zmienić hasło użytkownika, gdy z jakiś powodów użytkownik nie może zrobić tego samodzielnie. Administrator wybiera użytkownika z listy i dwukrotnie wpisuje nowe hasło, co przedstawia rysunek 19. Następnie osobiście lub telefonicznie przekazuje to hasło danemu użytkownikowi.

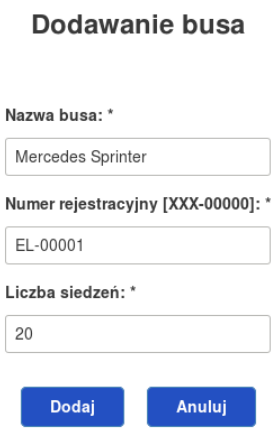


Rysunek 18: Zmień dane dowolnego konta

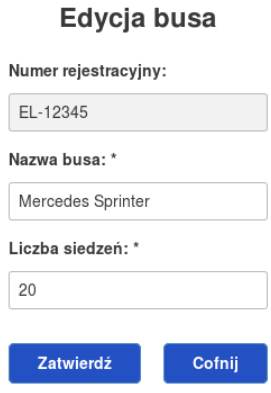


Rysunek 19: Zmień hasło dowolnego konta

Dodaj busa – przypadek użycia dostępny dla poziomu dostępu „Planista”. Po wypełnieniu formularza widocznego na rysunku 20 wymaganymi danymi i naciśnięciu przycisku „Dodaj”, dane zostają zapisane w bazie danych w odpowiednich kolumnach.



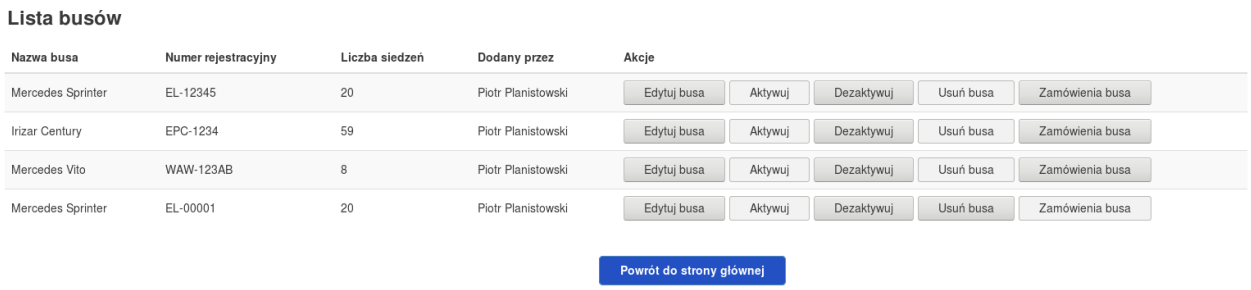
Rysunek 20: Dodaj nowego busa



Rysunek 21: Edytuj dane busa

Edytuj dane busa – umożliwia zmianę nazwy busa oraz liczby dostępnych w nim siedzeń poprzez formularz przedstawiony na rysunku 21. Edycja jest możliwa tylko w przypadku gdy bus nie jest wykorzystywany w żadnych aktualnych zamówieniach.

Wyświetl listę busów - powoduje pobranie z bazy i wyświetlenie w formie listy pokazanej na rysunku 22 wszystkich dostępnych busów. W tym miejscu możliwe jest wybranie opcji edycji danych busa, aktywowanie lub dezaktywowanie go lub usunięcie, jeżeli nie jest on powiązany z żadnym zamówieniem. Możliwe jest także przejście do listy, która przedstawia wszystkie zamówienia na wybranego z listy busa, opcja ta jest zablokowana w przypadku gdy bus nie jest przypisany do żadnych zamówień.

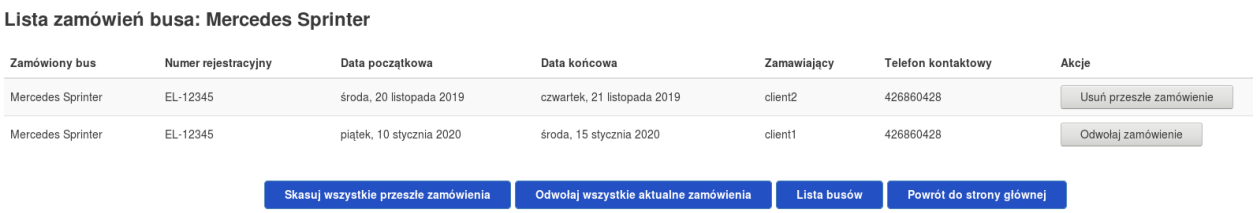


Rysunek 22: Wyświetl listę busów

Aktywuj/Dezaktywuj busa - dezaktywacja uniemożliwia wybór busa przez klienta, aż do momentu jego ponownej aktywacji. Blokada ta działa działa natychmiastowo, nawet gdy klient był już w trakcie składania zamówienia i wybrał busa z listy widocznej na rysunku 28. Dezaktywacja jest możliwa nawet, jeżeli bus posiada powiązania z istniejącymi zamówieniami. Wciśnięcie przycisku powoduje jego zablokowanie, działa on naprzemiennie z przyciskiem „Aktywuj". Na rysunku 22 przedstawiono cztery aktywne busy.

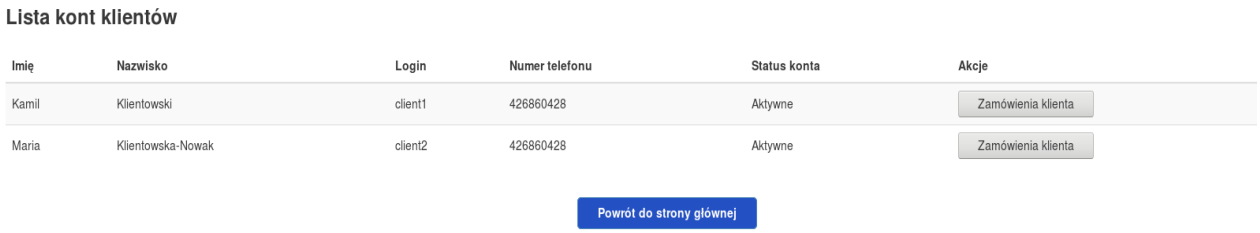
Usuń busa – poprzez naciśnięcie przycisku „Usuń busa" widocznego na rysunku 22 umożliwia planiście usunięcie busa z bazy, jeżeli tylko nie jej on powiązany z żadnym z zamówień.

Wyświetl listę zamówień busa - powoduje wyświetlenie lity wszystkich zamówień wybranego busa co prezentuje rysunek 23, umożliwia odwoływanie aktualnych i usuwanie przeszłych zamówień, również masowo poprzez przyciski dostępne na dole strony.



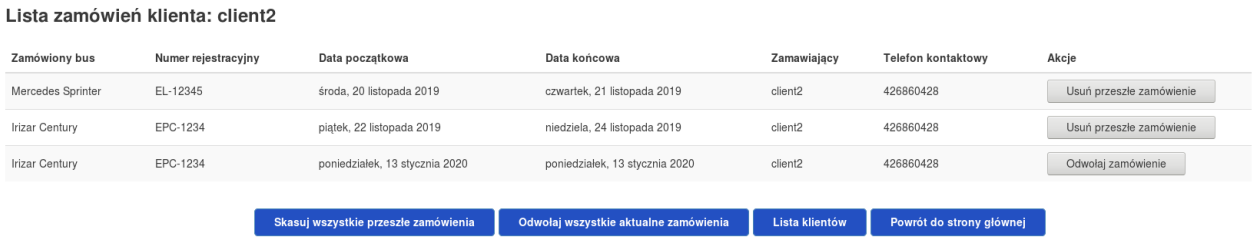
Rysunek 23: Wyświetl listę zamówień busa

Wyświetl listę klientów – wyświetla listę klientów wraz z ich wybranymi danymi widocznymi na rysunku 24. Umożliwia wyświetlenie listy wszystkich zamówień wybranego klienta.



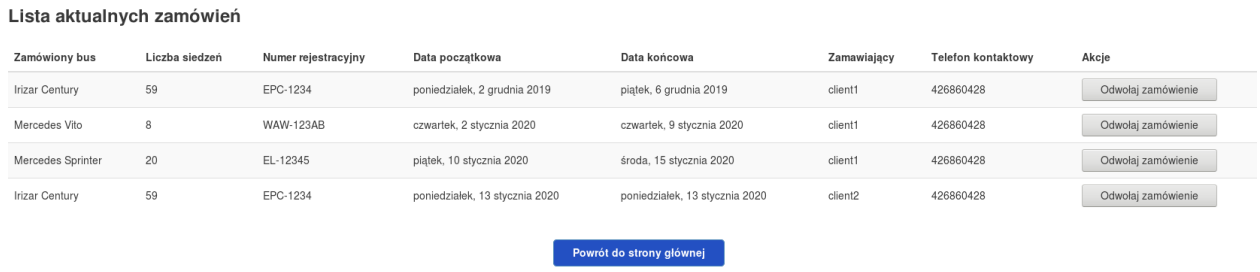
Rysunek 24: Wyświetl listę klientów

Wyświetl listę zamówień klienta – wyświetla listę wszystkich zamówień klienta, zaprezentowaną na rysunku 25, umożliwia odwoływanie aktualnych i usuwanie przeszłych zamówień, również masowo poprzez przyciski dostępne na dole strony.



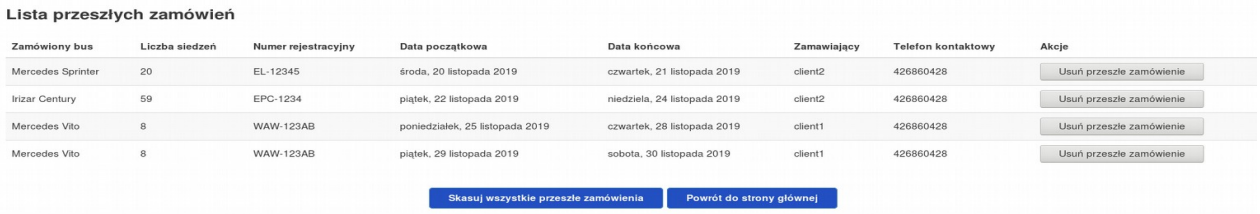
Rysunek 25: Wyświetl listę zamówień klienta

Wyświetl listę wszystkich aktualnych zamówień - ten przypadek użycia jest dostępny dla planisty i polega on na pobraniu z bazy i wyświetleniu wszystkich aktualnych zamówień, złożonych przez klientów, co zaprezentowano na rysunku 26. Możliwe jest tylko odwołanie pojedynczego zamówienia.



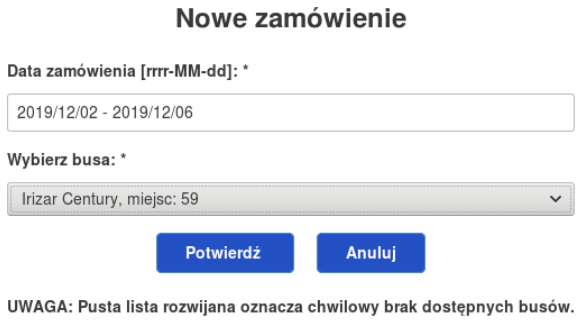
Rysunek 26: Wyświetl listę wszystkich aktualnych zamówień

Wyświetl listę wszystkich przeszłych zamówień - polega na pobraniu z bazy i wyświetleniu w formie listy widocznej na rysunku 27, wszystkich przeszłych zamówień. Ta lista daje możliwość usuwania zarówno pojedynczych zamówień jak i wszystkich przeszłych zamówień jednocześnie.

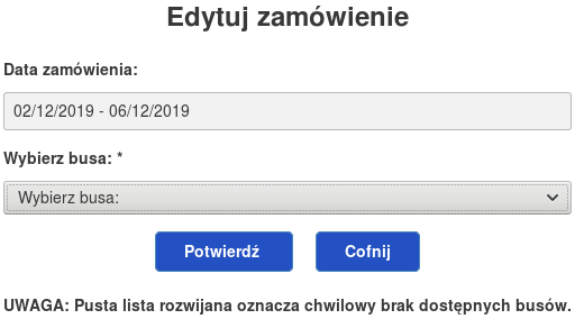


Rysunek 27: Wyświetl listę wszystkich przeszłych zamówień

Utwórz zamówienie - funkcjonalność dla poziomu dostępu „Klient”. Aby złożyć zamówienie wypożyczenia busa, należy najpierw wybrać zakres dat za pośrednictwem interaktywnego kalendarza, a następnie wybrać busa z listy rozwijanej. By potwierdzić złożenie zamówienia, należy kliknąć przycisk „Potwierdź”, co prezentuje rysunek 28.



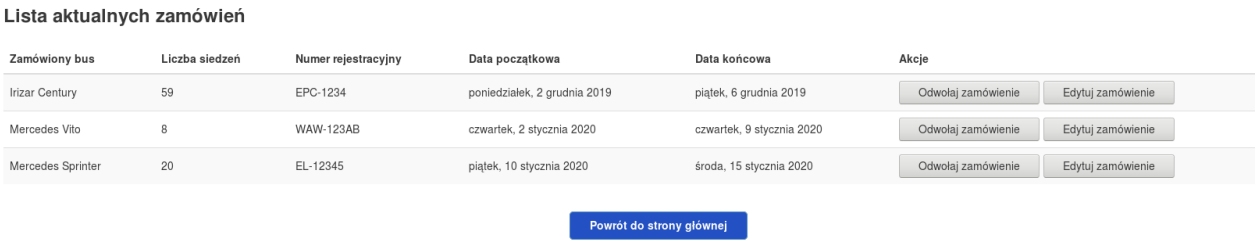
Rysunek 28: Utwórz zamówienie



Rysunek 29: Edytuj zamówienie

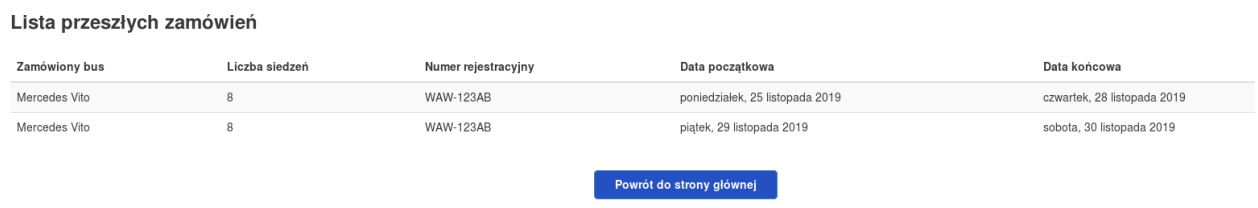
Edytuj zamówienie – jest opcją dostępną wyłącznie dla klienta. Umożliwia zmianę zamówionego busa w wybranym wcześniej terminie, na innego dostępnego z listy widocznej na rysunku 29.

Wyświetl listę swoich aktualnych zamówień - ten przypadek użycia polegający na wyświetleniu listy wszystkich aktualnych zamówień jest dostępny dla użytkowników z poziomem dostępu klient. Zaprezentowana na rysunku 30 lista, daje klientowi możliwość usunięcie pojedynczego zamówienia, oraz jego edycję.



Rysunek 30: Wyświetl listę swoich aktualnych zamówień

Wyświetl listę swoich przeszłych zamówień - pobiera z bazy i wyświetla wszystkie przeszłe zamówienia danego klienta co prezentuje rysunek 31, klient nie ma możliwości usuwania, ani edytowania żadnych przeszłych zamówień.



Rysunek 31: Wyświetl listę swoich przeszłych zamówień

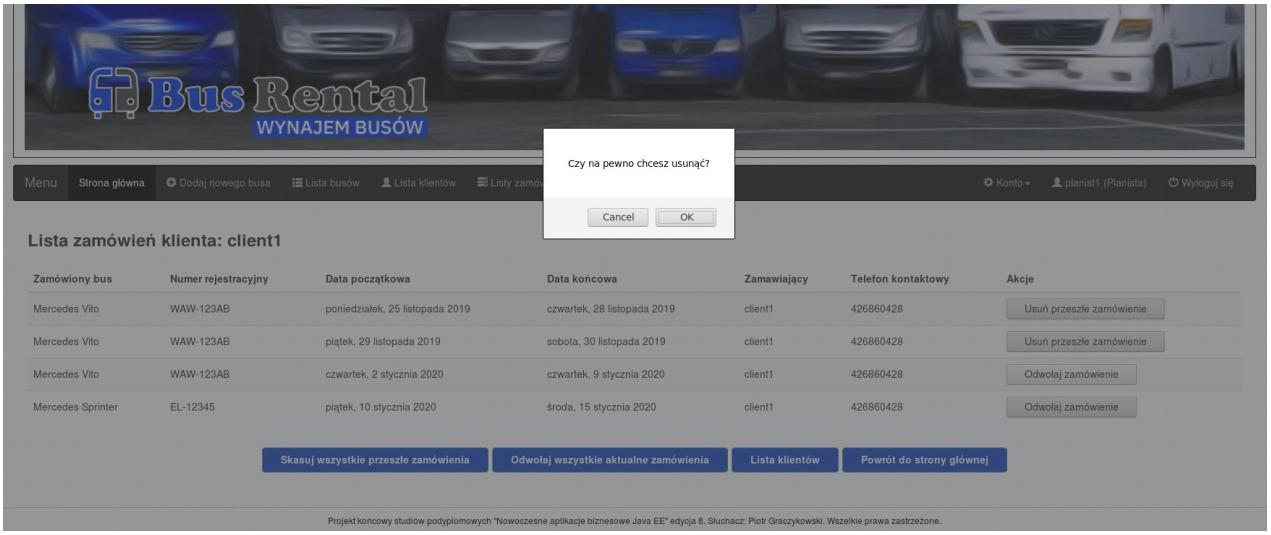
Odwołaj swoje aktualne zamówienie – umożliwia klientowi usunięcie pojedynczego, aktualnego zamówienia. Opcja ta jest dostępna dla klientów na liście ich aktualnych zamówień pokazanej na rysunku 30. Przed ostatecznym usunięciem zamówienia użytkownik musi potwierdzić swój wybór w oknie zabezpieczającym widocznym na rysunku 32.

Odwołaj dowolne aktualne zamówienie – umożliwia planiście odwołanie dowolnego, aktualnego zamówienia. Opcja ta jest dostępna dla planistów na listach zaprezentowanych na rysunkach 23, 25, 26. Przed ostatecznym usunięciem zamówienia użytkownik musi potwierdzić swój wybór w oknie zabezpieczającym widocznym na rysunku 32.

Usuń dowolne przeszłe zamówienie – pozwala planiście na usunięcie dowolnego, przeszłego zamówienia. Opcja ta jest dostępna dla planistów na listach zaprezentowanych na rysunkach 23, 25, 27. Przed ostatecznym usunięciem zamówienia użytkownik musi potwierdzić swój wybór w oknie zabezpieczającym widocznym na rysunku 32.

Odwołaj wszystkie aktualne zamówienia – dzięki tej funkcji użytkownik ma możliwość usunięcia wszystkich przeszłych rezerwacji. Opcja ta jest dostępna wyłącznie dla planistów na listach zaprezentowanych na rysunkach 23, 25. Przed ostatecznym usunięciem zamówień użytkownik musi potwierdzić swój wybór w oknie zabezpieczającym widocznym na rysunku 32.

Usuń wszystkie przeszłe zamówienia – dzięki tej funkcji użytkownik ma możliwość usunięcia wszystkich przeszłych rezerwacji. Opcja ta jest dostępna wyłącznie dla planistów na listach zaprezentowanych na rysunkach 23, 25, 27. Przed ostatecznym usunięciem zamówień użytkownik musi potwierdzić swój wybór w oknie zabezpieczającym widocznym na rysunku 32.



Rysunek 32: Okno potwierdzające akcję usuwania zamówień

3 Realizacja projektu

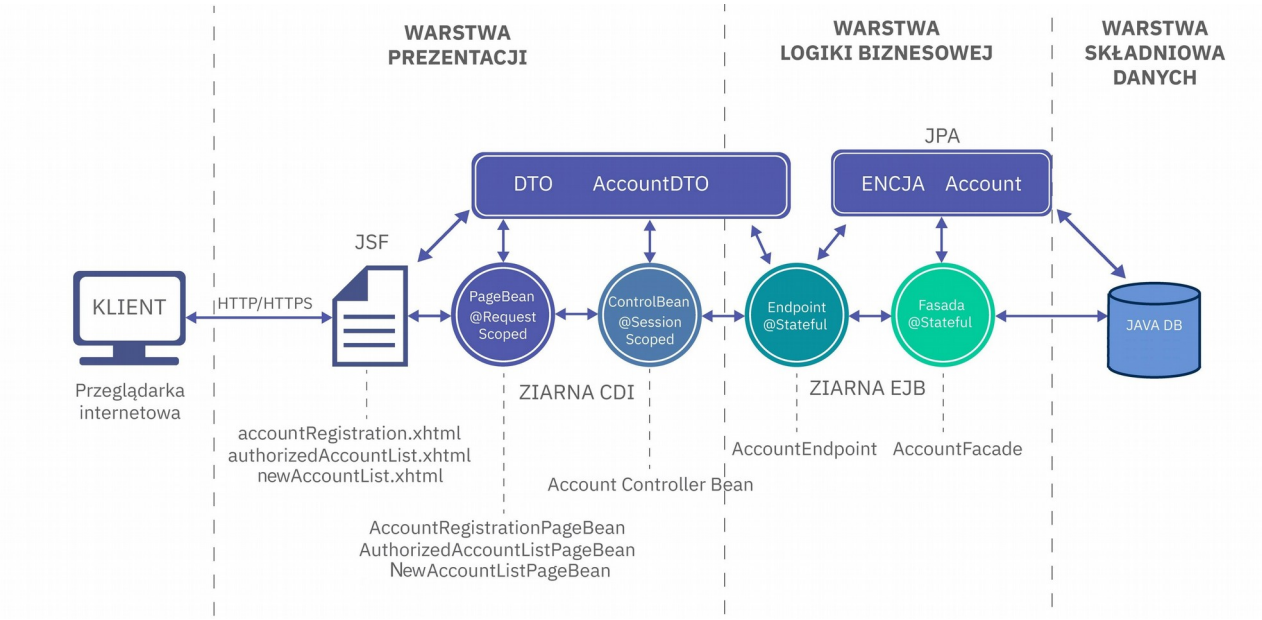
Aplikacja została zbudowana w architekturze trójwarstwowej [5], składającej się z warstwy składowania danych, warstwy logiki biznesowej oraz warstwy widoku. Wykorzystana architektura projektu zapewnia jego elastyczność i możliwość przyszłej zmiany jednego z jego komponentów.

3.1 Realizacja przykładowego CRUD'a

Bardzo ważnym etapem, podczas budowania aplikacji informatycznej jest dokładna analiza przepływu informacji w poszczególnych przypadkach użycia. Dzięki ich dokładnemu opisowi programista możne w każdej chwili weryfikować i modyfikować zaimplementowane rozwiązania, ponieważ w klarowny sposób obrazują one sposób w jaki dane użytkownika zostają przenoszone do i z bazy danych.

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

Poniżej znajduje się opis realizacji przykładowego CRUD dla kont użytkowników, wraz z odpowiadającym mu diagramem komponentów znajdującym się na rysunku 33.



Rysunek 33: Diagram komponentów dla przypadków użycia CRUD związanych z kontami użytkowników.

3.1.1 Tworzenie

Zarejestrowanie konta - Celem tego przypadku użycia jest utworzenie nieautoryzowanego konta użytkownika poprzez dodanie nowej pozycji w tabeli `ACCOUNT` w bazie danych. W tym celu nieuwierzytelniony użytkownik aplikacji wypełnia widoczny na rysunku 5 formularz znajdujący się na stronie *newAccount*.xhtml. Następnie, po wciśnięciu przycisku „Zarejestruj", zostaje wywołana metoda `newAccountAction` znajdująca się w klasie `NewAccountPageBean`.

```
<ui:composition template="./mainTemplate.xhtml">
 <ui:define name="content" class="flex-center flex-column">
   <h:form id="RegisterForm" class="text-left reservation-form">
      <h:messages globalOnly="true" styleClass="error_large" />
      <h:messages id="success" for="success" styleClass="confirm_large" />
       <div class="flex-column">
       <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="${newAccountPageBean.accountDTO.name}" >
        <f:validateLength minimum="2" maximum="60"  ></f:validateLength>
        <f:validateRegex pattern="[A-ZĄĆĘŁŃÓŚŹŻ][a-ząćęłńóśźż]+([\s][A-
      ZĄĆĘŁŃÓŚŹŻ][a-ząćęłńóśźż]+)?" />
   </h:inputText>
   <h:message for="name" styleClass="error_small"/> (...)
   <h:inputSecret class="margin-bottom-small" id="password"  maxlength="30"
      required="true"
      validatorMessage="#{msg['page.account.validator.password']}"  value="$
      {newAccountPageBean.accountDTO.password}" >
         <f:validateLength minimum="8" maximum="30"  ></f:validateLength>
         <f:validateRegex pattern="(?=.*\d)(?=.*[a-ząćęłńóśźż])(?=.*[A-
      ZĄĆĘŁŃÓŚŹŻ])(?=.*[\W]).{8,30}"/>
  </h:inputSecret>
  <h:message for="password" styleClass="error_small"/> (...)
       <div class="flex-center margin-top-small">
       <div>
           <h:commandButton class="main-button margin-right"
      value="#{msg['page.action.register']}" action="$
      {newAccountPageBean.newAccountAction()}" />
       </div>
```

Listing 1: Fragment strony newAccount.xhtml - formularz tworzenia konta użytkownika.

Metoda `newAccountAction` w klasie `NewAccountPageBean`, jest odpowiedzialna za sprawdzenie poprawności danych wprowadzonych w formularzu. W jej ciele wywoływana jest metoda `newAccount` klasy kontrolera `AccountControllerBean`, która jako parametr przyjmuje obiekt klasy `AccountDTO`. Metodę pokazuje listing 2.

```
@Named(value = "newAccountPageBean")
@RequestScoped
public class NewAccountPageBean implements Serializable {

    @Inject
    private AccountControllerBean accountControllerBean;
(…)
    public String newAccountAction() {
```

```
        if (passwordRepeat.equals(accountDTO.getPassword())) {
            try {
                accountControllerBean.newAccount(accountDTO);
            } catch (AppBaseException ex) {

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

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

Listing 2: Fragment kodu klasy AccountRegistrationPageBean.

Metoda `newAccount` klasy `AccountControlerBean` pokazana 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 zastosowano mechanizm, który weryfikuje czy nie doszło do odwołania realizowanej transakcji aplikacyjnej. Jeśli transakcja została odwołana następuje ponowne wykonanie metody. Gdy odwołanie nastąpi trzy razy zostaje zgłaszany wyjątek

```
@Named(value = "accountControllerBean")
@SessionScoped
public class AccountControllerBean implements Serializable {
    @EJB
    private AccountEndpoint accountEndpoint;
(...)

    public void newAccount(final AccountDTO accountDTO) throws
AppBaseException {
        final int UNIQ_METHOD_ID = accountDTO.hashCode() + 1;
        if (lastActionMethod != UNIQ_METHOD_ID) {
            int endpointCallCounter =
accountEndpoint.NB_ATEMPTS_FOR_METHOD_INVOCATION;
            do {
                accountEndpoint.registerAccount(accountDTO);
                endpointCallCounter--;
            } while (accountEndpoint.isLastTransactionRollback() &&
endpointCallCounter > 0);
            if (endpointCallCounter == 0) {
                throw
AppBaseException.creatExceptionForRepeatedTransactionRollback();
            }
            ContextUtils.emitI18NMessage("success", "error.success");
        } else {
            ContextUtils.emitI18NMessage(null, "error.repeated.action");
        }
        ContextUtils.getContext().getFlash().setKeepMessages(true);
        lastActionMethod = UNIQ_METHOD_ID;
    }
```

Listing 3: Fragment klasy AccountControlBean, metoda newAccount

Metoda `newAccount` klasy `AccountEndpoint` o widoczna na listingu 4 jest odpowiedzialna jest za przeniesienie danych z obiektu DTO na obiekt Encji i wywołanie metody `create` w klasie `AccountFacade`.

```
@PermitAll
public void newAccount(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 , metoda newAccount

W klasie `AccountFacade` następuje przeciążenie metody `create` widocznej na listingu 19 przy pomocy referencji do klasy nadrzędnej `super` wywołanie jej w nadrzędnej klasie abstrakcyjnej `AbstractFacade`. Metoda ta jako typizowany parametr przyjmuje obiekt klasy encyjnej. Następnie w klasie `AbstractFacade`, na obiekcie klasy `EntityMenager`, wywoływana jest metoda `persist`, pokazana na listingu 18, która powoduje utworzenie nowej krotki w tabeli `ACCOUNT` w bazy danych.

3.1.2 Wyświetlanie

Wyświetlenie listy nieautoryzowanych kont – przypadek użycia odpowiedzialny za pobranie z bazy danych listy nieautoryzowanych kont i wyświetleniu ich w formie tabeli w widoku użytkownika z poziomem dostępu **Administrator**, co pokazuje rysunek 16.

Za wyświetlenie odpowiednio sformatowanych danych odpowiedzialna jest strona widoczna na listingu 5 list*NewAccounts.xhtml* i zmienna `dataModelAccounts`, które przedstawiają w formie tabeli zwrócone z bazy dane.

```
<ui:composition template="./mainTemplate.xhtml">
  <ui:define name="content">
    <h:form id="AccountsForm" class="text-left reservation-form" >
       <div class="margin-bottom">
         <h:outputLabel style="font-size: x-large" class="margin-bottom"
      value="#{msg['page.new.accounts.list.title']}" />
         <h:messages globalOnly="true" styleClass="error_large" />
         <h:messages id="success" for="success" styleClass="confirm_large" />
           <div class="flex-column">
             <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>
(...)
            <f:facet
      name="header">${msg['page.header.label.actions']}</f:facet>
              <div>
               <h:selectOneMenu class="margin-right-small"
      value="#{row.accessLevel}" >
                  <f:selectItem itemLabel="$
      {msg['page.new.accounts.list.select.access.level']}"  />
                  <f:selectItems  value="$
      {listNewAccountsPageBean.listAccessLevels}" var="accessLevel"
      itemValue="#{accessLevel}"
      itemLabel="#{accessLevel .accessLevelI18NValue}" />
               </h:selectOneMenu>
                   <h:commandButton value="$
      {msg['page.new.accounts.list.action.set.access.level']}" class="margin-
      right-small" action="$
      {listNewAccountsPageBean.setAccessLevelForSelectedAccountAction(row)}"/>
                  <h:commandButton value="$
      {msg['page.new.accounts.list.action.delete']}" action="$
      {listNewAccountsPageBean.deleteSelectedAccountAction(row)}" onclick="if
      (! confirm('${msg['page.info.confirm.delete.action']}?') ) { return
      false;}; return true; "/>
              </div>
                 </h:column>
            </h:dataTable>
```

Listing 5: Fragment kodu strony listNewAccount.xhtml, wyświetlane dane, przycisk delete.

Klasa l`listNewAccountPageBean` jest odpowiedzialna za wywołanie metody inicjującej `initListNewAccounts` zaprezentowanej na listingu 6. Metoda ta przetwarza dane pobrane z pokazanej na listingu 7 klasy `AccountControlerBean` i przekazuje je stronie JSF poprzez obiekt klasy `DataModel`

```
@Named(value = "listNewAccountsPageBean")
@ViewScoped
public class ListNewAccountsPageBean {
(...)
    @PostConstruct
    public void initListNewAccounts() {
        try {
            listNewAccounts = accountControllerBean.listNewAccounts();
        } catch (AppBaseException ex) {
       Logger.getLogger(ListNewAccountsPageBean.class.getName())
      .log(Level.SEVERE, null, ex);
            ContextUtils.emitI18NMessage(null, ex.getMessage());
        }
        dataModelAccounts = new ListDataModel<>(listNewAccounts);

        AccessLevel[] listAllAccessLevels = AccessLevel.values();
        for (AccessLevel accessLevel : listAllAccessLevels) {
       accessLevel.setAccessLevelI18NValue(ContextUtils
      .getI18NMessage(accessLevel.getAccessLevelKey()));
        }
```

```
        listAccessLevels = new
      ArrayList<>(Arrays.asList(listAllAccessLevels));
        listAccessLevels.remove(AccessLevel.ACCOUNT);
        listAccessLevels.remove(AccessLevel.NEWACCOUNT);
    }

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

Listing 6: Fragment klasy ListNewAccountsPageBean, metoda inicjująca oraz metoda deleteAccountAccount

W metodzie zastosowano mechanizm, który weryfikuje czy nie doszło do odwołania realizowanej transakcji aplikacyjnej. Jeśli transakcja została odwołana następuje ponowne wykonanie metody. Gdy odwołanie nastąpi trzy razy zostaje zgłaszany wyjątek. W klasie klasy `AccountControlerBean` wywoływana jest metoda klasy punktu dostępowego `listNewAccounts`.

```
    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.creatExceptionForRepeatedTransactionRollback();
        }
        return selectedAccountsListsDTO;
    }
```

Listing 7: Fragment klasy AccountOrderControlerBean, metoda listNewAccounts

W dalszej kolejności dane pobierane są dzięki pokazanej na listingu 8 metodzie `initListNewAccounts` klasy `AccountEndpoint`, odpowiada ona za wywołanie metody `findNewAccount` klasy `AccountFacade` i przepisanie zwróconych przez nią danych w formacie obiektu encji na obiekt typu DTO.

```
    @RolesAllowed({"Admin"})
    public List<AccountDTO> listNewAccounts() throws AppBaseException {
        List<Account> listNewAccount = accountFacade.findNewAccount();
        savedAccountStateList = listNewAccount;
        List<AccountDTO> listNewRegisteredAccount = new ArrayList<>();
        for (Account account : listNewAccount) {
            AccountDTO accountDTO = new AccountDTO(
                    account.getLogin(),
                    account.getName(),
```

```
                account.getSurname(),
                account.getPhoneNumber(),
                account.getCreatedAt()
            );
            listNewRegisteredAccount.add(accountDTO);
        }
        Collections.sort(listNewRegisteredAccount);
        return listNewRegisteredAccount;
    }
```

Listing 8: Fragment kodu klasy AccountEndpoint, metoda listNewAccount

Metoda `findNewAccount` wywołuje kwerendę wyszukującą `Account.findNewAccount` w widocznej na listingu 19 klasie `AccountFacade` i przekazuje zwróconą przez nią dane w formie listy do klasy `AccountEndpoint`. Kwerenda `Account.findNewAccount` jest odpowiedzialna za pobranie z bazy danych z tabeli `ACCOUNT` wszystkich nieautoryzowanych kont użytkowników. Widoczna na listingu 17 kwerenda pobiera wszystkie konta z klasy `NewAccount`.

3.1.3 Edycja

Aktywowanie konta – przypadek użycia odpowiedzialny za aktualizowanie danych w bazie. Za pośrednictwem tej funkcji zmieniany jest status aktywności konta na aktywne (ang. *true*). Można się uwierzytelnić, jedynie na aktywne konto. Przycisk do aktywacji znajduje się na widocznej na rysunku 17 tabeli znajdującej się na stronie list*AuthorizedAccounts.xhtml* dostępnej z poziomu dostępu **Administrator.**

W celu aktywacji konta, należy wybrać w tabeli odpowiedni przycisk „Aktywuj konto", którego kod widoczny jest na listingu . Wywołuje on metodę `activateAccountAction` znajdującaj się w klasie `AuthorizedAccountsListPageBean`

```
<h:commandButton value="${msg['page.authorized.accounts.list.active']}"
class="button margin-right-small"
action="#{listAuthorizedAccountsPageBean.activateAccountAction(row)}"
disabled="#{row.active}" />
```

Listing 9: Fragment kodu strony listAuthorizedAccount.xhtml reprezentujący przycisk aktywacji konta

Wywołana metoda widoczna na listingu 10 przekazuje do kolejnej metody klasy `AccountControllerBean` wskazany jako parametr obiekt klasy `AccountDTO`. Po aktualizacji danych w bazie następuje odświeżenie widoku tabeli.

```
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; }
```

Listing 10: Fragment kodu klasy ListAuthorizedAccountPageBean odpowiedzialny za przekazanie wskazanego obiektu do aktywacji.

Metoda `activeAccount` klasy `AccountControlerBean` 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 były ponownie aktualizowane w bazie. W przypadku pozytywnej weryfikacji następuje wywoływanie metody `activeAccount` z parametrem `accountDTO.getLogin` znajdującej się w klasie `AccountEndpoint` zaprezentowanej na listingu 11. W metodzie zastosowano także mechanizm, który weryfikuje czy nie doszło do odwołania realizowanej transakcji aplikacyjnej. Jeśli transakcja została odwołana następuje ponowne wykonanie metody. Gdy odwołanie nastąpi trzy razy zostaje zgłaszany wyjątek

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

Listing 11: Fragment klasy AccountControlerBean , metoda acttivateAccount

W klasie `AccountEndpoint` następuje wywołanie metody fasady `findByLogin(login)`, która zwraca obiekt encji z obecnym stanem aktywności konta. Jeśli konto nie było aktywne, to ta właściwość zostaje do niego przypisywana poprzez ustawienie parametru `setActive(true)` i wywołanie metody drugiej metody fasady `accountFacade.edit` wraz z informacją, który administrator wprowadził tą zmianę. Metodę tą prezentuje listing 12.

```
    @RolesAllowed({"Admin"})
    public void activateAccount(AccountDTO accountDTO) throws AppBaseException
{
        Account account = selectAccountWithIterator(accountDTO.getLogin(),
      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.createExceptionAccountAlreadyActvivated(account);
        }
    }
```

Listing 12: Fragment kodu klasy AccountEndpoint, metoda activateAccount

W klasie `AccountFacade` wywoływana jest metoda widoczna na listingu 19 wyszukująca odpowiednią pozycje w bazie danych przy pomocy podanego w jej parametrze loginu. Następnie pozycja ta jest zwracana do klasy `AccountEndpoint`. Kwerenda `Account.findByLogin` przedstawiona na listingu 17 jest odpowiedzialna za pobranie z bazy danych z tabeli `ACCOUNT` danych konta o wskazanym loginie.

3.1.4 Usuwanie

Usuń konto - powoduje usunięcie z bazy danych wybranego z listy konta. Usuwanie dostępne jest w formie przycisku wyłącznie na widocznej na rysunku 16 stronie *listNewAccounts.xhtml*, ponieważ możliwość kasowania jest dopuszczalna tylko dla nieautoryzowanych kont. Dostęp do tej funkcji ma **Administrator**.

W celu skasowania konta, należy wybrać z tabeli odpowiedni przycisk „Usuń konto”, którego kod zaprezentowano na listingu 5. Przycisk odpowiada za wywołanie metody `deleteSelectedAccountAction` klasy `ListNewAccountsPageBean`.

Na listingu 6 zaprezentowana jest metoda `deleteSelectedAccountAction`, odpowiedzialna za przekazanie do metody kontrolera klasy `AccountControllerBean`, wskazanego jako parametr, obiektu klasy `AccountDTO`.

Metoda `activeAccount` klasy `AccountControlerBean` widoczna na listingu 13 ma za zadanie sprawdzić, czy nie jest wywoływana dwukrotnie ta sama czynność, 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`. W metodzie zastosowano mechanizm, który weryfikuje czy nie doszło do odwołania realizowanej transakcji aplikacyjnej. Jeśli transakcja została odwołana następuje ponowne wykonanie metody. Gdy odwołanie nastąpi trzy razy zostaje zgłaszany wyjątek

```
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.creatExceptionForRepeatedTransactionRollback();
            }
            ContextUtils.emitI18NMessage("AccountsForm:success",
      "error.success");
        } else {
            ContextUtils.emitI18NMessage(null, "error.repeated.action");
        }
        ContextUtils.getContext().getFlash().setKeepMessages(true);
        lastActionMethod = UNIQ_METHOD_ID;
    }
```

Listing 13: Fragment klasy AccountControlerBean , metoda deleteAction

W pokazanej na listingu 14 metodzie klasy `AccountEndpoint` następuje wywołanie metody klasy fasady `AccountFacade findByLogin`, która zwraca obiekt encji konta. Następnie obiekt ten jest przekazany jako parametr w widocznej na listingu 19 metodzie `remove` wywoływanej na tej samej fasadzie.

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

Listing 14: Fragment kodu klasy AccoutEndpoint , metoda deleteAccount

W klasie `AccountFacade` następuje przeciążenie metody `remove` i wywołanie jej w nadrzędnej klasie abstrakcyjnej `AbstractFacade`. Metoda ta jako typizowany parametr przyjmuje obiekt klasy encyjnej. Klasa fasady pokazana została na listingu 19. W klasie `AbstractFacade`, przedstawionej na listingu 18 na obiekcie klasy `EntityMenager`, wywoływana jest metoda `remove`, która powoduje usunięcie z tabeli `ACCOUNT` w bazie danych wybranego przez nas konta.

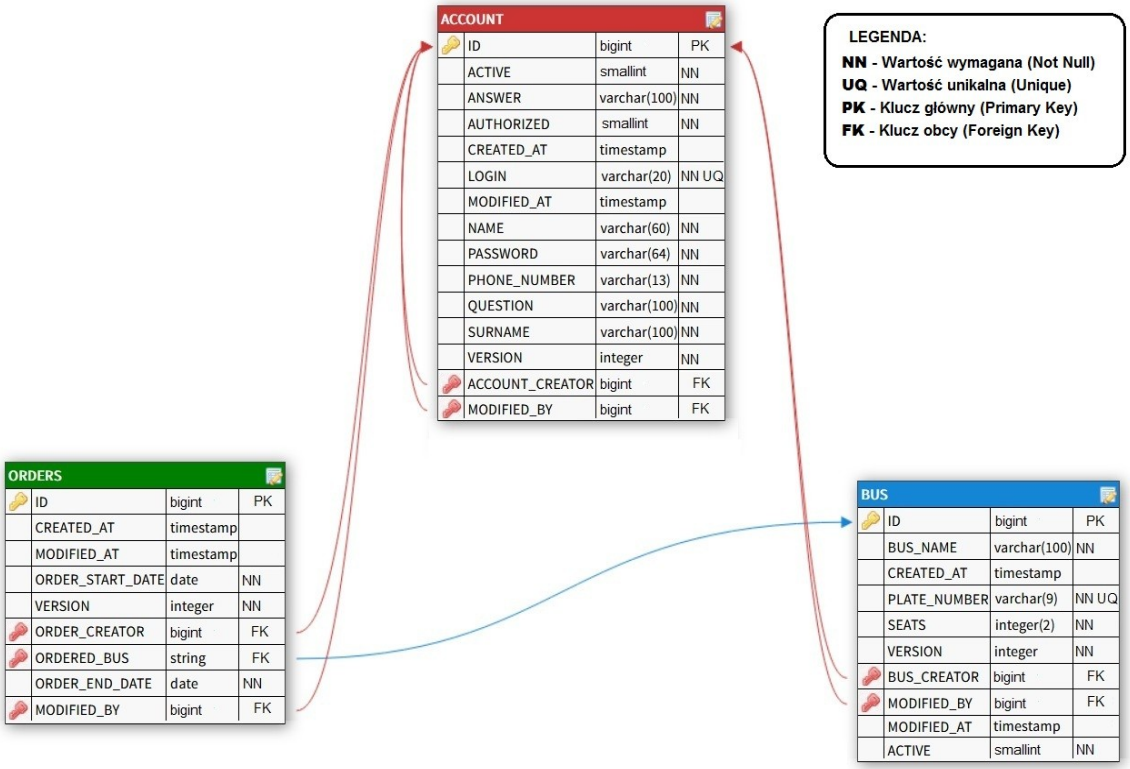
3.2 Warstwa składowania danych

Składowanie danych aplikacji odbywa się z wykorzystaniem relacyjnej bazy danych JavaDB. Jej struktury składają się z tabel danych powiązanych ze sobą za pomocą unikalnych pól kluczy głównych i pól kluczy obcych poszczególnych tabel [4]. Wartość klucza głównego nowego rekordu generowana jest z wykorzystaniem generatora tabel `@TableGenerator` przedstawionego w listingu 17. Wszystkie dane, za których wprowadzanie do bazy danych jest odpowiedzialny użytkownik aplikacji są wymagane (ang. *not null*) [4].

3.2.1 Model relacyjnej bazy danych

Rysunek 34 przedstawia diagram klas encyjnych, który obrazuje strukturę i powiązania występujące między obiektami w prezentowanym modelu danych. Model danych uwzględnia encje wykonane w standardzie JPA (Java Persistence API). Ponadto wykorzystano również

język zapytań JPQ służący do wyrażania zapytań operacyjnych na obiektach (encjach) [3]. Warunek wyszukiwania np. `SELECT a FROM Account a WHERE a.authorized = true`, zaprezentowany w listingu 17 pozwala na zawężenie zbioru wyników do kont posiadających parametr aktywności o wartości prawda (ang. *true*). Należy jednak pamiętać, że w języku JPQ odwołujemy się jedynie do elementu encji, a nie do tabel bazy danych.



Rysunek 34: Relacyjny model bazy danych

3.2.2 Konfiguracja zasobów w relacyjnej bazie danych

Za konfigurację zasobów w relacyjnej bazie danych odpowiada deskryptor składowania presistance.xml, który został zaprezentowany na listingu 15. Definiuje on jednostkę składowania wykorzystującą zasób JDBC [5]. Do połączenia z bazą danych aplikacja korzysta z puli połączeń JDBC [3] (ang. *JDBC Connection Pool*) i powiązanego z nią zasobu JDBC [14](ang. *JDBC Resource*) widocznych na listingu 44.

```
<?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="BusRentalPU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/BusRentalDS</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 15: Plik deskryptora składowania persistence.xml

3.2.3 Mapowanie obiektowo-relacyjne ORM

Aplikacja działa na modelu obiektowym opartym na klasach Javy i powiązanych z nimi relacyjnych tabelach bazy danych. Relacyjne bazy danych swój mechanizm działania wspierają na relacjach. Co za tym idzie relacja to zbiór rekordów, a każdy z nich, może zawierać atrybuty określonych typów. Stosując jednak bardziej popularną nomenklaturę jest możliwe powiedzenie, że relacje w relacyjnych bazach danych to tabele mieszczące w sobie dane. Założeniem mechanizmu ORM [14] (ang. *Object Relational Mapping*) jest więc powiązanie ze sobą konkretnych obiektów, zawierających zarówno proste dane, jak i odwołania do innych obiektów, z rekordami występującymi w obrębie tabel (relacji) [14].

Do utworzenia mapowania między obiema strukturami (klasami Javy i tabelami bazy danych) została wykorzystana wspomniana wyżej specyfikacja Java Persistance API, która odpowiada za prawidłowe odwzorowanie struktur baz danych przy użyciu odpowiedniej adnotacji `@Column` znajdującej się w poszczególnych klasach encyjnych, których przykłady jprzedstawiają w listingi 16 i 17.

Klasa `AbstractEntity` zaprezentowana w listingu 16, to nadrzędna klasa encyjna implementująca część mechanizmu blokad optymistycznych, poprzez deklarację pola `Version` oraz mapująca wspólne dla wszystkich klas pola `createdAt i modifiedAt` na odpowiadające im kolumny `CREATED_AT i MODIFIED_AT` w relacyjnej bazie danych. Pola te zapewniają informację o dacie utworzenia i ostatniej modyfikacji poszczególnych 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 16: Klasa AbstractEntity - Implementacja bazowej klasy dla encji

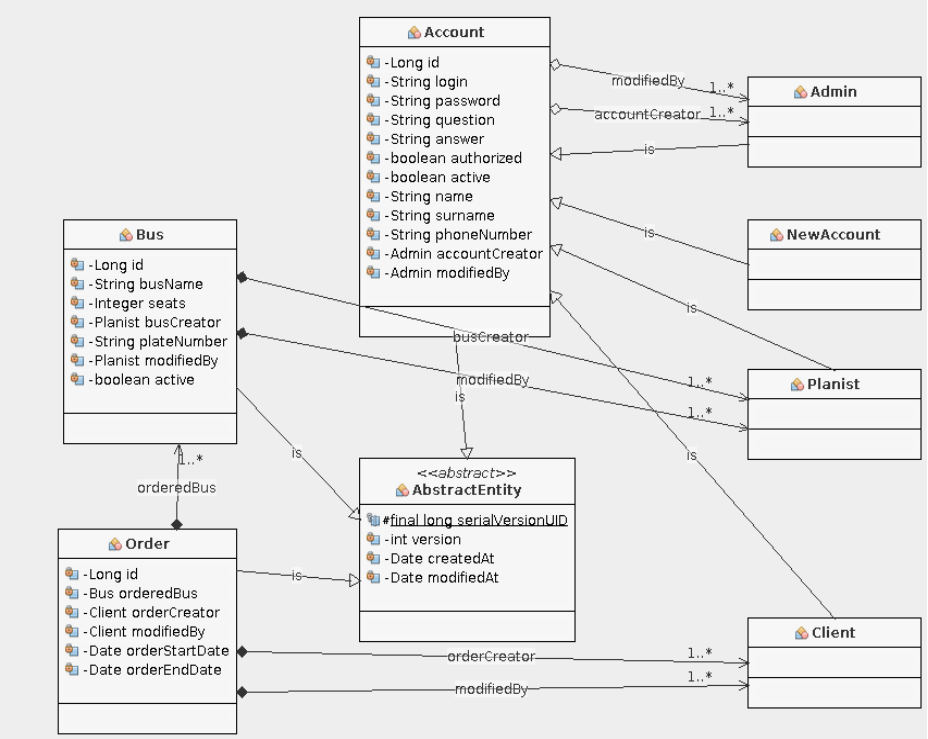
Na listingu 26 zaprezentowano fragment klasy encyjnej `Account`. Klasa ta, wykorzystując strategię dziedziczenia pojedynczej tabeli (ang. Single table) [3], łączy poprzez kolumnę dyskryminatora (adnotacja `@DiscriminatorColumn`) o nazwie `LEVEL_OF_ACCESS` klasy `Admin, Planist, Client, NewAccount`. Dzięki temu rozwiązaniu klasa Account rozszerza pięć klas encyjnych, a obiekty tych klas są odwzorowane 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.findClientAccounts", query = "SELECT a FROM
      Client 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 17: Fragment klasy encyjnej Account.

Rysunek 35 przedstawia diagram klas encyjnych, który obrazuje strukturę i powiązania występujące między obiektami w prezentowanym modelu danych. Model danych uwzględnia encje wykonane w standardzie JPA (Java Persistence API).



Rysunek 35: Diagram klas encyjnych.

3.3 Warstwa logiki biznesowej

Przy tworzeniu warstwy biznesowej aplikacji została wykorzystana technologia Java EE, komponenty sesyjne EJB i kontener EJB . Zadaniem warstwy logiki biznesowej jest przetwarzanie otrzymanych i zwracanych danych według ustalonego modelu biznesowego.

3.3.1 Sesyjne komponenty EJB

Sesyjne komponenty EJB zostały podzielone na dwie grupy, pierwsza z nich zawiera komponenty bezstanowe z adnotacją `@Stateless`, które reprezentują fasady (ang. facade i zawierają metody realizujące podstawowe operacje na rekordach przechowywanych w strukturach relacyjnej bazy danych. Klasą nadrzędną wszystkich fasad jest widoczna na listingu 18, klasa `AbstractFacade` udostępniająca podstawowe metody do komunikacji z bazą danych. Wszystkie klasy dziedziczące z klasy `AbstractFacade` nadpisują jej metody i uzupełniają je o występujące w danym przypadku użycia wyjątki, co ukazane jest na przykładzie fragmentu klasy `BusFacade` zaprezentowanej na listingu 19.

```
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 18: Fragment klasy abstrakcyjnej AbstractFacade.

```
@Stateless
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public class AccountFacade extends AbstractFacade<Account> {

    static final public String DB_UNIQUE_CONSTRAINT_ACCOUNT_LOGIN =
      "UNIQUE_LOGIN";
    static final public String DB_FK_BUS_BUS_CREATOR = "FK_BUS_BUS_CREATOR";
    static final public String DB_FK_BUS_MODIFIED_BY = "FK_BUS_MODIFIED_BY";
    static final public String DB_FK_ACCOUNT_MODIFIED_BY =
      "ACCOUNTMODIFIED_BY";
    static final public String DB_FK_ACCOUNT_ACCOUNT_CREATOR =
      "CCUNTCCOUNTCREATOR";
    static final public String DB_FK_ORDERS_MODIFIED_BY =
      "ORDERS_MODIFIED_BY";
    static final public String DB_FK_ORDERS_ORDER_CREATOR =
      "ORDERSORDERCREATOR";

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

    @Override
    protected EntityManager getEntityManager() {
        return em;
    }
    public AccountFacade() {
        super(Account.class);
    }
    @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);
            }
        }
    }
```

```
@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 (OptimisticLockException e) {
        throw AppBaseException.createExceptionOptimisticLock(e);
    } catch (PersistenceException e) {
        throw AppBaseException.createExceptionDatabaseQueryProblem(e);
    }
}
@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_BUS_BUS_CREATOR) ||
  cause.getMessage().contains(DB_FK_BUS_MODIFIED_BY))) {
            throw AccountException.createExceptionAccountInUseInBus(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_ORDERS_ORDER_CREATOR) ||
  cause.getMessage().contains(DB_FK_ORDERS_MODIFIED_BY))) {
            throw AccountException.createExceptionAccountInUseInOrder(e,
  entity);
        } else {
            throw AppBaseException.createExceptionDatabaseQueryProblem(e);
        }
    }
}
@PermitAll
public Account findByLogin(String login) throws AppBaseException {
    TypedQuery<Account> tq = em.createNamedQuery("Account.findByLogin",
  Account.class);
    tq.setParameter("login", login);
    try {
        return tq.getSingleResult();
```

```
        } catch (NoResultException e) {
            throw AccountException.createExceptionAccountNotFound(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);
            }
        }
    }
    @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 19: Fragment klasy AccountFacade.

Druga grupa to komponenty stanowe `@Statefull`, które reprezentowane są przez punkty dostępowe (ang. *endpoint*) umożliwiające komunikację warstwy widoku z warstwą logiki biznesowej. Są one odpowiedzialne za transfer danych z obiektów DTO [12] (and. Data Transfer Object) do obiektów encji JPA, co zwiększa poziom bezpieczeństwa przechowywanych w bazie danych. Ponadto obiekty DTO przechowują tylko informacje potrzebne dla danego przypadku. Sposób przenoszenia danych między obiektem DTO i obiektem encyjnym zaprezentowano na przykładzie metody `newBus()` klasy `BusEndpoint` widocznej na listingu 21.

3.3.2 Mechanizmy ochrony spójności danych

Aby zachować spójność danych w aplikacji został zaimplementowany system przetwarzania transakcyjnego (ang. *On-Line Transactional Processing*). Komponent EJB przy pomocy zaimplementowanego interfejsu `javax.ejx.SessionSynchronization` udostępnia kontenerowi metody zwrotne, widoczne na listingu 20 przedstawiającym klasę `AbstractEndpoint`. Umożliwiają one zapis danych oraz identyfikacje realizowanych transakcji aplikacyjnych.

```
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 20: Klasa abstrakcyjna AbstractEndpoint

W aplikacji zastosowano strategię zarządzania transakcjami przez kontener, tzw. strategię CMT (ang. *Container-Management Transactions*). Transakcje rozpoczynają się w klasach punktu dostępowego, dzięki atrybutowi `TransactionAttributeType.REQUIRES_NEW`, co zostało ukazane na przykładzie klasy `BusEndpoint` której fragment przedstawia listing 21, następnie kontekst transakcji aplikacyjnej trafia do klasy fasady, która posiada atrybut `TransactionAttributeType.MANDATORY` widoczny na listingu 19 klasy Account`Facade`.

```
@Stateful
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
@Interceptors(LoggingInterceptor.class)
public class BusEndpoint extends AbstractEndpoint implements
SessionSynchronization {

    @EJB
    private PlanistFacade planistFacade; (...)
    private Bus busState; (...)
    private List<Bus> savedBusStateList;
    public Bus getBusState() {
        return busState;
    }
    public void setBusState(Bus busState) {
        this.busState = busState;
    }
    @RolesAllowed({"Planist"})
    private Planist loadCurrentPlanist() throws AppBaseException {
        String planistLogin = sessionContext.getCallerPrincipal().getName();
        Planist planistAccount = planistFacade.findByLogin(planistLogin);
        if (planistAccount == null) {
            throw AppBaseException.createExceptionNotAuthorizedAction();
        }
        if (!planistAccount.isActive()) {
            throw
AccountException.createExceptionAccountNotActive(planistAccount);
```

```
        }
        return planistAccount;
    } (...)
    @RolesAllowed({"Planist"})
    public void newBus(BusDTO busDTO) throws AppBaseException {
        if (busDTO.getSeats() >= 8 && busDTO.getSeats() < 61) {
            Bus bus = new Bus();
            bus.setBusName(busDTO.getBusName());
            bus.setPlateNumber(busDTO.getPlateNumber());
            bus.setSeats(busDTO.getSeats());
            bus.setActive(true);
            bus.setBusCreator(loadCurrentPlanist());
            busFacade.create(bus);
        } else {
            throw BusException.createExceptionBusWrongSeatsNumber();
        }
    }
```

Listing 21: Fragment klasy BusEndpoint

Podczas równoległego przetwarzania transakcji w wielodostępnym systemie, może zdarzyć się sytuacja, w której dojdzie do konfliktu podczas zapisu i odczytu danych z bazy. Niezbędne jest zatem zastosowanie odpowiedniego poziomu izolacji transakcji. Zastosowany poziom izolacji transakcji o wartości `read-committed` przedstawia listing 44. Za pośrednictwem mechanizmu blokad optymistycznych, istnieje możliwość sprawdzenia, czy zapisane w bazie dane nie uległy zmianom od momentu ich odczytu z bazy. Kontroluje to `EntityManager` poprzez porównanie pola z numerem wersji, oznaczonego adnotacją `@Version`. Pole wersji jest zdefiniowane w klasie abstrakcyjnej `AbstractEntity`, co zaprezentowano na listingu 16. Obiekt, który został pobrany z bazy danych jest zapamiętywany w polu stanowego komponentu EJB. Gdy wystąpi niezgodność wersji tego obiektu z wersją obiektu aktualnie znajdującego się w bazie, zostanie zgłoszony widoczny na listingu 24 wyjątek `OptimisticLockException`, którego obsłużenie zapewnia metoda `edit` oraz `delete` w fasadzie, widoczna na listingu 19.

3.3.3 Kontrola dostępu

W logice biznesowej, dla poszczególnych ról zdefiniowano kontrolę dostępu oraz uprawnienia do wykonywania określonych czynności. W rozdziale 3.4.4 Uwierzytelnianie i autoryzacja opisano sposób odwzorowanie kont użytkowników na role, dostarczony za pośrednictwem mechanizmu uwierzytelniania. Korzystając z pakietu `javax.annotation.security` i adnotacji, które on oferuje, dostęp do biznesowych metod w punktach dostępowych orazfasadach został przyznany określonym poziomom dostępu. Na listingach klasy fasady 19 <CRUD Fasada> oraz punktu dostępowego 21 oraz <CRUD Endpoint> zaprezentowano przykład wykorzystania adnotacji `@RolesAllowed`.

3.3.4 Kontrola odpowiedzialności

Kontrola odpowiedzialności w aplikacji obywa się dla przypadków użycia dotyczących tworzenia i edycji danych poprzez rejestrowanie w tabelach baz danych takich informacji jak data utworzenia, data modyfikacji oraz danych wprowadzających ich użytkowników

w postaci kluczy obcych. Wyjątek stanowi przypadek użycia dotyczący resetowania hasła dla konta, oraz edycja własnych danych przez użytkowników innych niż administratorzy, w tych wypadkach kolumny `MODIFIED_BY` poszczególnych tabel bazy danych nie mają ustawiane żadnej wartości (`null`). Jest to spowodowane zastosowaną w projekcie strategią bezpieczeństwa ograniczającą niektórym użytkownikom dostęp do najważniejszych danych i przyznawaniem uprawnień do wprowadzania zmian w kolumnach `MODIFIED_BY` tylko jednemu poziomowi dostępu.

Oprócz przechowywania danych w bazie danych, system rejestruje też podstawowe informacje o wykonywanych przez użytkowników akcjach w dziennikach zdarzeń, za co odpowiedzialny jest mechanizm przechwytujący znajdujący, zaprezentowany na listingu 22 klasy `LogginInterceptor`

```
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(" wystapil wyjatek " + ex);
            throw ex;
        } finally {
            Logger.getGlobal().log(Level.INFO, sb.toString());
        }
```

Listing 22: Fragment klasy LoggingInterceptor – mechanizm przechwytujący

3.3.5 Obsługa błędów

W aplikacji zaimplementowano wielopoziomowy system obsługi zgłaszanych przez nią wyjątków[7]. Wyjątki nieobsługiwane przez aplikację [16](tzn. wyjątki systemowe), które zgłasza kontener EJB prowadzą do odwołania bieżącej transakcji i wywołania strony błędu

zadeklarowanej w deskryptorze wdrożenia *web.xml* widocznym na listingu 23 [14]. Aby ujednolicić obsługę wyjątków w aplikacji dla wszystkich wyjątków stworzono nadklasę `AppBaseException`, której fragment zaprezentowano na listingu 24, z której dziedziczą klasy deklarujące bardziej szczegółowe wyjątki związane z przypadkami użycia danej sekcji aplikacji tj. `AccountException`, `BusException` i `OrderException`. Klasa nadrzędna opatrzona jest adnotacją `@ApplicationException(rollback = true)` [4], co powoduje, że mimo odwołania transakcji i propagacji wyjątku kontener nie stosuje standardowej procedury obsługi wyjątku, a komponent EJB nie zostanie zniszczony. Poszczególne metody w kolejnych warstwach aplikacji zapewniają propagację wyjątku, dzięki zgłoszeniu go w sygnaturach ich metod (`throws`). Zgłoszony wyjątek jest przekazywany do warstwy widoku aplikacji, gdzie następuje wyświetlenie odpowiedniego komunikatu użytkownikowi, informującego go o rodzaju zgłoszonego wyjątku i wymaganej od niego reakcji. Obsługę wyjątku w warstwie widoku aplikacji opisuje rozdział 3.4.6 Obsługa błędów .

```
(...)
<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 23: 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_DATABASE_CONNECTION_PROBLEM =
      "error.database.connection.problem";
    static final public String KEY_NOT_AUTHORIZED_ACTION =
      "error.not.authorized.account.problem";
(...)
    protected AppBaseException(String message, Throwable cause) {
        super(message, cause); (...)
    }
    public static AppBaseException
      createExceptionOptimisticLock(OptimisticLockException e) {
        return new AppBaseException(KEY_OPTIMISTIC_LOCK);
    }
    public static AppBaseException
      createExceptionDatabaseConnectionProblem(Throwable e) {
        return new AppBaseException(KEY_DATABASE_CONNECTION_PROBLEM);
    }
```

```
    public static AppBaseException createExceptionNotAuthorizedAction() {
        return new AppBaseException(KEY_NOT_AUTHORIZED_ACTION);
    }(...)
```

Listing 24: Fragment klasy wyjątków AppBaseException.

3.4 Warstwa widoku

Warstwa widoku w aplikacji opiera się głównie na szkielecie (ang. *framework*) JSF (ang. *JavaServer Faces*)[1] i języku HTML (ang. *HyperText Markup Language*)[17], które są obecne we wszystkich jej stronach. Dla ujednolicenia widoku stron została wykorzystana technologia Bootstrap[17], co pozwoliło na podział widoku na cztery obszary: nagłówek, pasek menu, zawartość główna, oraz stopka. Bardziej zaawansowane funkcjonalności, takie jak interaktywny kalendarz czy potwierdzenie usunięcia danych z tabeli, zostały zaimplementowane w wykorzystaniem języka JavaScript[18][17].

3.4.1 Wzorzec projektowy DTO

W aplikacji zastosowano model danych DTO [12], czyli Obiekt Transferu Danych (ang. *Data Transfer Object*), który daje możliwość odseparowania od siebie danych przekazywanych między warstwą składowania danych, a warstwą widoku. DTO przenosi dane między warstwą widoku i warstwą logiki biznesowej, gdzie wewnątrz klas punktu dostępowego współpracuje z obiektami klas encyjnych i przepisuje na nie lub z nich wybrane dane. Dzięki takiemu rozwiązaniu użytkownik nigdy nie ma dostępu do krytycznych dla aplikacji danych, takich jak `Version` lub `ID`, jak również do danych nieistotnych do obsługi interfejsu użytkownika dla danego przypadku użycia, a ważnych z punktu widzenia bezpieczeństwa jak np. `password` lub dane dotyczące kontroli odpowiedzialności np. `accountCreator` lub `modifiedBy`. Przykładem klasy DTO jest `BusDTO`, której fragment prezentuje listing 25.

```
public class BusDTO implements Comparable<BusDTO> {

    private String busName;
    private String plateNumber;
    private int seats;
    private boolean active;
    private Planist busCreator;
    private boolean activeReservations;
    private Date createdAt;
    public BusDTO() {
    }
    public BusDTO(String busName, String plateNumber, int seats) {
        this.busName = busName;
        this.plateNumber = plateNumber;
        this.seats = seats;
    }
    (...)
        @Override
    public String toString() {
        return busName + ", " + ContextUtils.printI18NMessage("bus.dto.seats")
      + ": " + seats;
    }
    @Override
    public int compareTo(BusDTO o) {
```

```
        return this.createdAt.compareTo(o.createdAt);
    }
```

Listing ,25: Fragment klasy BusDTO

Dodatkową funkcjonalnością klas DTO jest zaimplementowany interfejs `Comparable`, który za pomocą metody `compareTo` widocznej na listingu 25, daje możliwość porównywania obiektów po jego określonych cechach i zwracania ich w formie posortowanej listy. Sortowanie list odbywa się z użyciem metody `sort` klasy `Collections` do przedstawiono na przykładzie listingu 8 klasy punktu dostępowego.

3.4.2 Ujednolicony interfejs użytkownika

Wszystkie strony aplikacji posiadają ujednolicony interfejs graficzny niezależnie od poziomu dostępu użytkownika i zostały wykonane przy pomocy technologii JSF, HTML i CSS (ang. Cascading Style Sheets), które umożliwiają odpowiednie przystosowanie wyglądu stron dzięki dostarczanymi z nimi bibliotekami. JSF udostępnia różnorodne wzorce szablonów JSF, jeden z nich, którego kod przedstawiono na listingu 26, został wykorzystany do podzielenia wszystkich stron warstwy widoku na cztery osobne sekcje, poprzez polecenie `<ui:insert name=" ">`: nagłówek (`header`) widoczny na rysunku 36, pasek menu (`menu`) zaprezentowany na rysunkach 37, 38, 39 i 40, zawartość główną (`content`) oraz stopkę (`footer`) ukazane na rysunku 41. Strony korzystające z szablonu mogę implementować całą domyślną zawartość sekcji szablonu dzięki poleceniu `<ui:composition template="./mainTemplate.xhtml">` i wprowadzać zmiany wyłącznie w pożądanych sekcjach, co zaprezentowano na listingu 27, przedstawiającym fragment kodu strony `mainPage.xhtml` zmieniającym jedynie sekcję `<ui:define name="content">`.

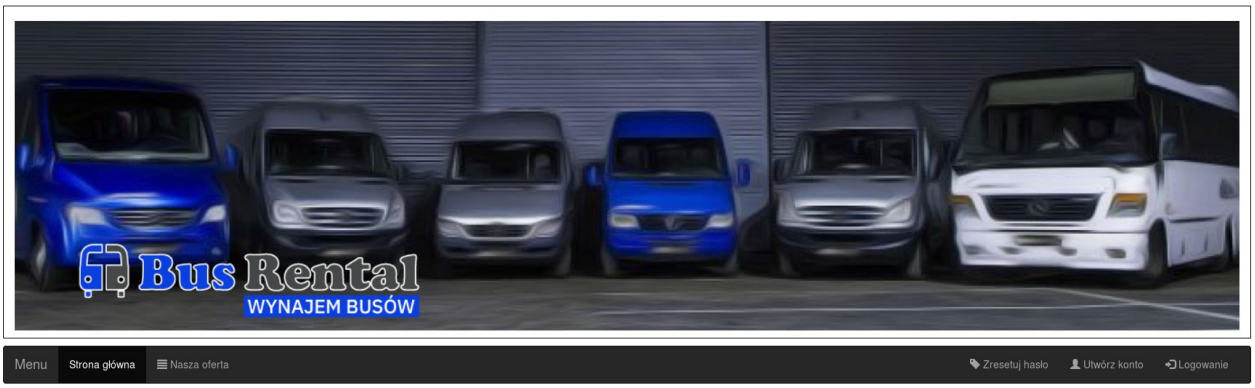
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">
    <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"/>
        <title>#{msg['page.main.template.title']}</title>
        <h:outputStylesheet name="/css/daterangepicker.css" />
    </h:head>
    <h:body>
       <div class="container-fluid ">
          <div class="row">
            <div class="col-md-12 header" align="center">
                <ui:insert name="header">(...)</ui:insert>
            </div>
            <div class="col-md-12 menu">
                <ui:insert name="menu">(...)</ui:insert>
            </div>
            <div class="col-md-12 content-area">
```

```
                <ui:insert name="content">(...)</ui:insert>
            </div>
            <div class="col-md-12 footer">
                <ui:insert name="footer">(...)</ui:insert>
            (...)
        <h:outputScript name="/js/jquery.js"/>
        <h:outputScript name="/js/moment.min.js"/>
        <h:outputScript name="/js/daterangepicker.min.js"/>
        <h:outputScript name="/js/bootstrap.min.js"/>
        <h:outputScript name="/js/app.js"/>
    </h:body>
</html>
```

Listing 26: Fragment szablonu mainTemplate.xhtml z implementacją biblioteki Bootstrap

```
(...)<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 27: Fragment strony mainPage.xhtml



Rysunek 36: Wygląd sekcji nagłówka strony.

Rysunek 37: Wygląd paska menu dla użytkownika nieuwierzytelnionego.

Rysunek 38: Wygląd paska menu dla poziomu dostępu: Administrator

Rysunek 39: Wygląd paska menu dla poziomu dostępu: Planista

Rysunek 40: Wygląd paska menu dla poziomu dostępu: Klient

Nawigacja pomiędzy stronami odbywa się poprzez wybranie jednego z przycisków widocznego na przedstawionych na rysunkach 37, 38, 39 i40 paskach menu poszczególnych poziomów dostępu. Inną zaimplementowaną w aplikacji formą nawigacji jest wykorzystanie poleceń `<navigation-case>` w pliku konfiguracyjnym *faces-config.xml*, którego fragment pokazano na listingu 28. Zawarte w nim reguły nawigacji są wykorzystywane w ziarnach CDI typu `PageBean` poprzez metody łańcuchów znaków `String`, które w zwracanej wartości zawierają daną regułę nawigacji.

Rysunek 41: Wygląd sekcji zwartości głównej (content) na przykładzie strony mainPage.xhtml i sekcji stopki (footer).

3.4.3 Internacjonalizacja

System obsługuje dwie wersje językowe: polską i angielską. Obsługiwane języki, wraz z językiem domyślnym, ustalone są na podstawie pliku `faces-config.xml`, którego fragment przedstawiony jest na listingu 28. Wybór języka odbywa się automatycznie na podstawie ustawień w opcjach przeglądarki internetowej, z której korzysta użytkownik systemu.

```
(...)<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>main</from-outcome>
    <to-view-id>/mainPage.xhtml</to-view-id>
    <redirect />
</navigation-case> (...)
```

Listing 28: Fragment pliku faces-config.xml

Aplikacja wykorzystuje internacjonalizację zgodną ze standardem i18n widocznym na listingu 29 przedstawiającym plik *web.xml*. Wyświetlenie właściwego komunikatu na stronie odbywa się przy użyciu klucza i przypisanej do niego wartości, wyszczególnionych na listingu 30. Zbiór tych wartości znajduję się zestawie plików `messages.properties`.

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

Listing 29: Fragment pliku web.xml – ustawienia internacjonalizacji

```
# accountRegistration.xhtml
page.register.title=Rejestracja konta
page.register.form.label.name= Imię
page.register.form.label.surname= Nazwisko
page.register.form.label.login= Login
page.register.form.label.password= Hasło
(...)
# Błędy
(...)
error.bus.exist.problem= Bus z podanym numerem rejestracyjnym już istnieje.
error.bus.wrong.state.problem= Wykryto niezgodność edytowanego busu.
error.bus.is.in.order.problem= Bus jest przypisany do aktualnych zamówień.
error.bus.already.active.problem= Bus został już wcześniej aktywowany.
error.bus.already.deactive.problem= Bus został już wcześniej dezaktywowany.
error.bus.unavailable.problem= Wybrany bus jest niedostępny.
error.bus.edited.problem= Bus był w międzyczasie edytowany. Wybierz ponownie.
error.bus.has.active.orders.problem= Nie można edytować busa z aktywnymi
zamówieniami.(...)
```

Listing 30: Fragment pliku messages_pl.properties

Aby skrót komunikatu wyświetlił jego właściwą dla wybranego języka wartość konieczna jest jego konwersja poprzez zadeklarowanie go na stronach JSF wewnątrz specjalnego wyrażenia, np. `#{msg[page.register.title]}`, powoduje to wywołanie metod klasy `ContexUtils`, której fragment zaprezentowano na listingu 31, która zwraca właściwy komunikat wyświetlany użytkownikowi.

```
(...)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 31: Fragmenty klasy ContextUtils.

3.4.4 Uwierzytelnianie i autoryzacja

Aby uwierzytelnić się w aplikacji użytkownik musi posiadać unikalny login i hasło przypisane do istniejącego, aktywnego konta i wpisać je w formularzu widocznym na rysunku 6, który wykorzystując polecenia widoczne na listingu 32 strony uwierzytelniania *login.xhtml*. Po udanym zalogowaniu użytkownik zostaje przeniesiony na stronę główną *mainPage.xhtml*, co zostało zadeklarowane w pliku konfiguracyjnym *web.xml*, którego fragment widoczny jest na listingu 33.

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

Listing 32: Fragment pliku login.xhtml

```
<login-config>
     <auth-method>FORM</auth-method>
     <realm-name>BusRentalJDBCRealm</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 33: 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 37, 38, 39 i 40 , fragment klasy *mainTemplate.xhtml* zawierający podział widoku został przedstawiony na listingu 34.

```
<f:subview id="unauthenticatedView" rendered="#{empty request.remoteUser}">
    <li><a href="ourOffer.xhtml">(...)
</f:subview>
<f:subview id="clientView" rendered="#{request.isUserInRole('Client')}">
    <li><a href="newOrder.xhtml">(...)
    <li><a href="listMyOrders.xhtml">(...)
</f:subview>
<f:subview id="planistView" rendered="#{request.isUserInRole('Planist')}">
```

Listing 34: Fragment pliku mainTemplate.xhtml przedstawiający podział w widoku menu

Deskryptora wdrożenia aplikacji *web.xml* jest odpowiedzialny za deklarację ról i ograniczeń dostępu do wybranych zasobów aplikacji. Fragment jego kodu wraz z ustawieniami wykorzystania protokołu HTTPS [2] (ang. *Hypertext Transfer Protocol Secure*), szyfrującego dane przy pomocy protokołu *SSL [2] (*ang. *Secure Socket Layer)* , został przedstawiony na listingu 35.

```
<security-constraint>
     <display-name>AuthenticatedUserPages</display-name>
     <web-resource-collection>
         <web-resource-name>AuthenticatedUserPages</web-resource-name>
         <description/>
         <url-pattern>/faces/viewMyAccount.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>Planist</role-name>
         <role-name>Client</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>Planist</role-name>
    </security-role>
    <security-role>
        <description/>
        <role-name>Client</role-name>
    </security-role>
```

Listing 35: Fragmenty kodu deskryptora wdrożenia web.xml

Deskryptor wdrożenia aplikacji *glassfish-web.xml* jest odpowiedzialny za mapowanie użytkowników i grup na odpowiadające im role, z których każda reprezentuje inny poziom dostępu. Jego fragment został zaprezentowany na listingu 36.

```
<security-role-mapping>
  <role-name>Admin</role-name>
  <group-name>access.level.admin</group-name>
</security-role-mapping>
<security-role-mapping>
  <role-name>Planist</role-name>
  <group-name>access.level.planist</group-name>
</security-role-mapping>
<security-role-mapping>
  <role-name>Client</role-name>
  <group-name>access.level.client</group-name>
</security-role-mapping>
```

Listing 36: 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 należy podać nazwę źródła danych uwierzytelniania zadeklarowaną wcześniej w deskryptorze wdrożenia w elemencie `<realm-name>`, widoczną na listingu 33. Konfiguracja serwera została pokazana na listingu 46.

Hasła wszystkich użytkowników, które wymagane są do uwierzytelnienia się w aplikacji internetowej osadzonej w serwerze Payara przechowywane są w bazie danych w postaci niejawnej. Trafiają tam jako skrót obliczony algorytmem SHA-256. Obliczanie skrótu hasła za pomocą jednokierunkowej funkcji odbywa się w widocznej na listingu 37 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 37: Fragment klasy encyjnej Account

3.4.5 Walidacja danych

Walidacja danych odbywa się dzięki walidatorom umieszczonym wewnątrz plików xhtml w warstwie widoku projektu, co pokazuje listing 38 prezentujący walidację danych dodawanego busa. Walidatory są wyrażeniami regularnymi i kontrolują format wprowadzanych przez użytkownika danych, tak aby był zgodny z formatem wymaganym przy wprowadzaniu ich do bazy danych.

```
(...)
<h:inputText class="margin-bottom-small" id="plateNumber" maxlength="9"
required="true" validatorMessage="#{msg['page.bus.validator.platenumber']}"
             value="${newBusPageBean.busDTO.plateNumber}" >
    <f:validateLength minimum="6" maximum="9"  />
    <f:validateRegex pattern="[A-Z]{2,3}\-[0-9A-Z]{4,5}" />
</h:inputText>
<h:message for="plateNumber" styleClass="error_small"/> (...)
```

Listing 38: Fragment pliku newBus.xhtml – walidacja wprowadzanych danych.

W razie potrzeby dane mogą zostać konwertowane na zmienne innej klasy, co pokazuje listing 39 ukazujący konwersję daty jako łańcucha znaków, na obiekt klasy `Date`.

```
(...)  public void setOrderStringDate(String orderStringDate) throws
ParseException {
        this.orderStartDate = new
SimpleDateFormat("yyyy/MM/dd").parse(orderStringDate.substring(0, 10));
        this.orderEndDate = new
SimpleDateFormat("yyyy/MM/dd").parse(orderStringDate.substring(13));
        this.orderStringDate = orderStringDate;
(...)
```

Listing 39: Fragment klasy OrderDTO – konwertowanie łańcucha znaków na klasę Date.

W przypadku błędnego wprowadzenia danych walidator wyświetla stosowny, internacjolizowany komunikat, widoczny na listingu 38 jako wartość parametru `validatorMessage`.

3.4.6 Obsługa błędów

Obsługa błędów w warstwie widoku odbywa się w klasach typu `PageBean` poprzez obsługę propagowanych do niej wyjątków aplikacyjnych dzięki deklaracjom `throws` występującym we wszystkich metodach pozostałych klas[6] czego przykład widzimy na listingach 19 lub 21 [10]. Klasa `PageBean` może również deklarować własne wyjątki w formie internacjonalizowanych komunikatów i wyświetlać je razem z wyjątkami aplikacyjnymi z użycie klasy `ContextUtils`. Czego przykładem jest metoda `registerAccountAction()` widoczna na listingu 40 klasy `newAccountPageBean`. Komunikaty te wyświetlają się na stronach JSF dzięki użyciu elementów `<h:messages(...)/>` widocznych na przykładzie strony *newAccount.xhtml* na listingu 41. Oprócz tego w warstwie widoku wywoływany jest L*ogger*, który odnotowuje wystąpienie wyjątku w dzienniku zdarzeń i zgłasza jego poziom [9].

```
 (...) 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 40: Fragment klasy NewAccountPageBean.

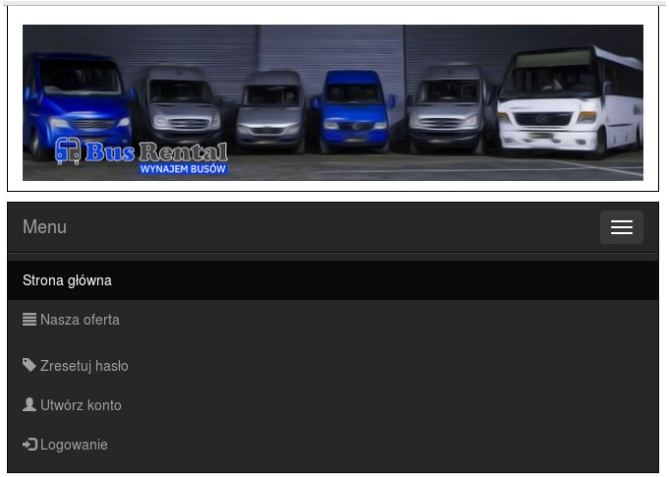
```
(...)
 <h:messages globalOnly="true" styleClass="error_large" />
 <h:messages id="success" for="success" styleClass="confirm_large" />
(...)
   <h:inputText class="margin-bottom-small" id="name" maxlength="60"
required="true" validatorMessage="#{msg['page.account.validator.name']}"
    value="${newAccountPageBean.accountDTO.name}" >
     <f:validateLength minimum="2" maximum="60"  ></f:validateLength>
   </h:inputText>
   <h:message for="name" styleClass="error_small"/> (...)
```

Listing 41: Fragment strony newAccount.xhtml

3.4.7 Technologie do obsługi interfejsu graficznego

Główną technologią wykorzystaną do utworzenia interfejsu graficznego jest JSF, którego wykorzystanie zostało szerzej opisane w rozdziale 3.4.2 Ujednolicony interfejs użytkownika. Utworzony dzięki niej szablon *mainTemplate.xhtml* widoczny na listingu 26 i korzystające z niego strony klientów szablonu, których fragmenty zostały zaprezentowane na listingach 1, 5 i 9 pozwalają zachować jednolity wygląd aplikacji i ułatwiają rozbudowywanie go o kolejne elementy.

Warstwa widoku wykorzystuje również bibliotekę CSS *Bootstrap,* która została wykorzystana do utworzenia responsywnego paska menu, widocznego na rysunkach 37, 38, 39 i 40, który po zmianie rozdzielczości lub rozmiaru okna strony odpowiednio dostosowuje swój wygląd, co prezentuje rysunek 42, fragment kodu strony *mainTemaple.xhtml* odpowiedzialny na generowanie paska menu znajduje się na listingu 42.

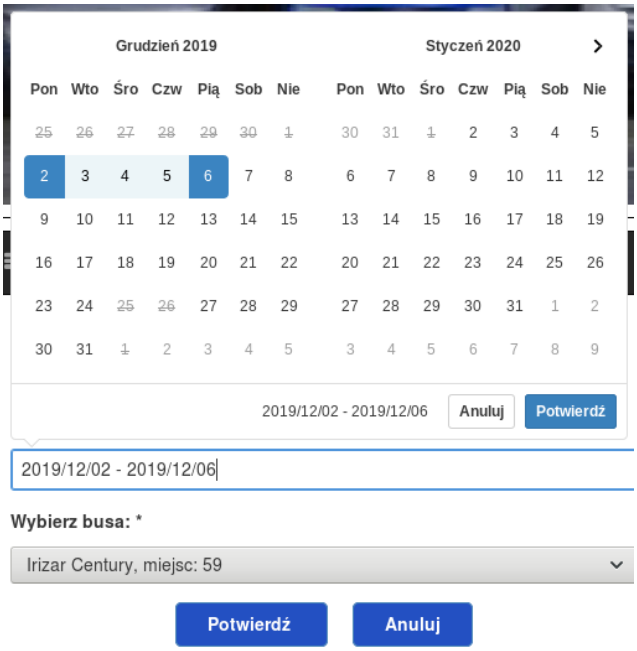


Rysunek 42: Responsywny pasek menu

```
<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 42: Fragment szablonu mainTemlate.xhtml odpowiedzialny za wyświetlenie paska menu

Do wyświetlenia interaktywnego kalendarza zaprezentowanego na rysunku 43 dla przypadku użycia dotyczącego utworzenia nowego zamówienia została wykorzystana biblioteka *JavaScript* o nazwie *Date Range Picker*, jej użycie deklarują polecenia `<h:outputStylesheet(…)/>` i `<h:outputScript(…)/>` widoczne w listingu 26 strony *mainTemaplete.xhtml*. Oraz polecenie `<h:inputText class="datepicker margin-bottom-small" id="orderDate"(…)/>` strony *newOrder.xhtml*.



Rysunek 43: Strona newOrder.xhtml - kalendarz

Warstwa widoku aplikacji została utworzona dzięki zastosowaniu ziaren CDI [8] (ang. *Contexts and Dependency Injection*), czyli klas o cyklu życia realizowanym przez kontener wstrzykiwania zależności. Ich cykl życia zależny jest od funkcji, jaką dana klasa pełni w aplikacji. Ziarna CDI obsługują żądania użytkowników systemu i przekazują wybrane dane biznesowe na między stronami xhtml i logiką biznesową aplikacji.

W projekcie zastosowano ziarna o zasięgu [14]: żądania `@RequestScoped` dla większości stron formularzy co pokazują listingi 40 i 2, zasięgu sesji `@SesionScoped` dla klas kontrolera służących do zapamiętywania i przekazywania danych do kolejnych żądań oraz komunikacji z warstwą logiki biznesowej, pokazanych na listingu 3, zasięgu widoku `@ViewScooped` dla klas wyświetlających listy, której przykład widzimy na listingu 6 w celu utrzymania stanu wyświetlonej listy, tak aby nie dopuścić do wybrania niepoprawnych danych z uwagi na wielodostęp, oraz zasięgu aplikacyjnym `@AplicationScoped` dla klasy *mainApplicationBean* widoczna na listingu 43, która jest odpowiedzialna za metody wyświetlające login i poziom dostępu w interfejsie użytkownika zbudowanego systemu.

```
@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 43: Fragment klasy MainApplicationPageBean

3.5 Instrukcja wdrożenia

Do uruchomienia stworzonego systemu niezbędne jest przygotowanie środowiska, które posiada działający system operacyjny oraz współczesną przeglądarkę internetową. Do tego należy posiadać zainstalowane i uruchomione: serwer aplikacyjny Payara w wersji 5.183 oraz system Java DB (*Apache Derby*) w wersji 10.13.1.1., a następnie:

1. Utworzyć bazę danych z użyciem programu narzędziowego `ij` dostarczonego wraz z Apache Derby. Listingu 44 przedstawia konfigurację bazy danych.

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

Listing 44: Fragment pliku konfiguracyjnego glassfish-resource.xml

Bazę danych o nazwie `jdbc:derby://localhost:1527/BusRental` należy utworzyć w katalogu z bazami danych zlokalizowanym w katalogu domowym używając polecenia `connect`, następnie ustawić `create` na `true`, jak zaprezentowano na listingu 45, dla domyślnego dla Derby katalogu `.netbeans-derby`.

2. Następnie należy, zgodnie z poleceniami pokazanymi na listingu 45, 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 BusRental*/src/main/resources/* *.

3. 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 44.

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

Listing 45: Tworzenie bazy danych JavaDB dla systemu operacyjnego Linux przy pomocy programu narzędziowego ij

4. Konfiguracja obszaru bezpieczeństwa na serwerze polega na otwarciu przeglądarki internetowej i wywołaniu adresu URL: *http://localhost:4848/common/index.jsf*

Z menu po lewej stronie należy wybrać klikając kolejno: *Configurations* → *server-config* → *Security* → *Realms*. Następnie należy wybrać przycisk *New* znajdujący się po prawej stronie. Otwarty formularz należy uzupełnić podanymi w listingu 46 wartościami i wycisnąć *OK*.

```
Realm Name: BusRentalJDBCRealm
Class Name: com.sun.enterprise.security.auth.realm.jdbc.JDBCRealm
JAAS Context: jdbcRealm
JNDI: jdbc/BusRentalDS
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
```

Listing 46: Dane do konfiguracji obszaru bezpieczeństwa na serwerze Payara

5. Aby wgrać aplikację na serwer aplikacyjny należy z menu po lewej stronie wybrać przycisk *Application*, następnie wybrać przycisk *Deploy.* W otwartym oknie, poprzez użycie *Browse* wybrać plik BusRental-*1.0.war,* który należy pobrać z załączonego do raportu nośnika. Do pliku prowadzi ścieżka *BusRental/target/BusRental-1.0.war*. Po wybraniu pliku należy wybrać przycisk *Otwórz* w okienku na dole. Następnie zatwierdzić wybór *OK*. Aplikacja powinna być widoczna na serwerze. Aby ją uruchomić należy wybrać z kolumny *Action* przycisk *Launch.*

Jeśli cała procedura była przeprowadzona poprawnie, użytkownik aplikacji uzyska możliwość zalogowania się na jedno z predefiniowanych kont, znajdujących się w pliku *initDB.sql*, ich poziomy dostępu, loginy i hasła zostały ujawnione na listingu 47.

```
Poziom dostępu administrator: login: admin1, hasło: Qwerty1!
Poziom dostępu planista: login: admin1, hasło: Qwerty1!
Poziom dostępu klient: login: admin1, hasło: Qwerty1!
```

Listing 47: Dane uwierzytelniania dla utworzonego systemu

3.6 Podsumowanie

Celem niniejszej pracy było stworzenie systemu informatycznego wspomagającego składanie rezerwacji na busy w firmie przewozowej. Procedura budowy aplikacji została podzielona na kilka etapów. Początkowo zostały opracowane założenia odnośnie funkcjonalności systemu oraz dokonano wyboru odpowiednich technologii. Następnym etapem było opracowanie struktur relacyjnej bazy danych a także implementacja aplikacji. Ostatnim krokiem było przygotowanie danych inicjujących i uruchomienie aplikacji internetowej w serwerze aplikacyjnym. Aplikacja realizuje wszystkie założenia i wymagania, które zostały postawione na początku dokumentu. Były to m. in.: obsługa błędów, autoryzacja, internacjonalizacja, uwierzytelnianie, walidacja, wielodostęp, ochrona spójności danych, blokady optymistyczne, przetwarzanie transakcyjne, rejestrowanie zdarzeń w dziennikach i mapowanie obiektowo-relacyjne.

System został wykonany w architekturze trójwarstwowej. W każdej z nich zostały zastosowane inne technologie. W warstwie widoku został wykorzystany szkielet JSF

oraz technologie i narzędzia takie jak CSS, JavaScript oraz bilbioteka Bootstrap. logika biznesowa została zaimplementowana przy użyciu ziaren EJB, a do komunikacji z warstwą widoku wykorzystuje obiekty transferowe DTO, co zwiększa bezpieczeństwo przechowywanych danych poprzez kontrolę nad tym, jakie dane trafiają do użytkownika aplikacji. Rekordy w bazie danych powstały z modelu wspieranego przez adnotacje mapowania obiektowo-relacyjnego w standardzie JPA.

Aplikacja została wykonana na platformie Java Enterprise Edition oraz osadzona na serwerze aplikacyjnym Payara. Wszystkie dane składowane są w trzech tabelach zarządzanych przez system Apache Derby. Przy tworzeniu aplikacji wykorzystano zintegrowane środowisko programistyczne NetBeans.

Stworzony system zapewnia cztery poziomy dostępu i funkcje użytkownika nieuwierzytelnionego. Użytkownik nieuwierzytelniony, czyli **Gość**, może przeglądać ofertę wypożyczalni busów, zalogować się, utworzyć nowe konto lub zresetować hasło. **Administrator** posiada możliwość zarządzania kontami wszystkich użytkowników, m. in. ich autoryzowania, edytowania, usuwania oraz aktywowaniem i dezaktywowania. **Planista** odpowiedzialny jest za obsługę zamówień na wypożyczenia busów, może przeglądać ich listy według różnych kryteriów, odwoływać aktualne i kasować przeszłe zamówienia. Zarządza też flotą busów, może je dodawać, usuwać, aktywować, dezaktywować oraz edytować ich dane. Ma też dostęp do listy klientów i przypisanych do nich zamówień. **Klient** ma możliwość utworzenia zgłoszenia na wypożyczenie autobusu poprzez wybór busa i okresu zamówienia poprzez interaktywny kalendarz. Ma dostęp do listy swoich aktualnych i przeszłych zamówień. Może edytować lub usuwać swoje aktualne zamówienia.

W aplikacji zaimplementowano szczegółowo opracowaną obsługę błędów. W celu zapobieganiu uszkodzenia danych, każdy zgłoszony wyjątek aplikacyjny wymusza wycofanie transakcji aplikacyjnej a jego obsługa odbywa się poprzez propagowanie go do warstwy widoku i poinformowaniu użytkownika o wystąpieniu błędu. W metodzie fabrykującej obiekt wyjątku następuje również ustawienie klucza komunikatu błędu dla użytkownika aplikacji. Dzięki zastosowanej w aplikacji internacjonalizacji klucz błędu jest tłumaczony i wyświetlony użytkownikowi w języku zgodnym z jego preferencjami. System domyślnie udostępnia interfejs użytkownika w języku zgodnym z ustawieniami przeglądarki internetowej użytkownika a obsługiwanymi językami są język polski oraz angielski. W aplikacji zastosowano strategię zarządzania transakcjami przez kontener, tzw. strategię CMT. Wszystkie transakcje biznesowe obsługiwane przez system poddane są mechanizmowi audytu. Do serwera aplikacji została delegowana obsługa uprawnień oraz kontroli dostępu. Dostęp do biznesowych metod w punktach dostępowych oraz fasadach został przyznany określonym poziomom dostępu za pomocą adnotacji *RollesAllowed*. W aplikacji zaimplementowano mechanizm rejestrujący podejmowane przez użytkowników czynności biznesowe w dziennikach zdarzeń. W dziennikach tych utrwalane są też informacje o granicach transakcji bazodanowych wraz ze statusem jej zakończenia.

Motywacją do stworzenia oprogramowania było własne doświadczenie w pracy w firmie przewozowej, która obecnie nie korzysta z żadnego systemu informatycznego. Firma ta skorzysta z ręcznie tworzonego grafiku, a przyjmowanie zamówień odbywa się telefonicznie lub za pośrednictwem poczty elektronicznej. Zastosowanie opracowanego systemu informatycznego znacznie przyśpieszy i zautomatyzuje działanie firmy, co odciąży zatrudnione w niej osoby. Klienci zyskają za to nowy sposób na składania zamówień i przeglądania oferty firmy.

Resumując, uważam, że cel pracy został osiągnięty. Stworzony system informatyczny spełnia zarówno wszystkie wymagania funkcjonalne, jak i niefunkcjonalne.

3.7 Perspektywy dalszego rozwoju programu

Uważam że stworzony system informatyczny jest kompletny i już w obecnej formie nadaje się do użytkowania w małej firmie transportowej, Jednak nadal chciałbym rozwijać go i dodawać do niego nowe funkcjonalności, wśród nich znajdują się:

Wprowadzenie nowego poziomu dostępu **Kierowcy** i zapewnienie klientom możliwości dodawania ich do zamówień.

Rozbudowa sekcji składania zamówień na wypożyczenia busów. Dodawanie ilości pasażerów, punkt odbioru przewożonych osób, miejsce docelowe podróży, możliwość zamawiania większej ilości busów.

Możliwość składania zamówień na busy co do godziny i minuty, wprowadzenie interaktywnego kalendarza ułatwiającego pracę planiście. Zwiększenie możliwości sortowania tabel z widokiem zamówień.

Dodatkowe tabele w bazie danych dające możliwość, np.: tabela zarządzania dniami wolnymi od pracy lub tabela historyczna przechowująca wybrane date przez określony okres czasu.

System mailingowy i smsowy, wspomagający pracę administratorów przy obsłudze kont, jak i planistów przy informowaniu klientów o realizacji zamówień.

4 Źródła

Bibliografia

1. Anghel Leonard: JavaServer Faces 2.2. Mistrzowskie programowanie, Wydanie , Helion, 2016
2. Basham Bryan, Sierra Kathy, Bates Bart: Head First. Servlets & JSP.. Edycja polska., Wydanie I, Helion, 2005
3. Baue Ch., King G., Gregory G.: Java Persistence. Programowanie aplikacji bazodanowych w Hibernate. Wydanie II, Wydanie , Helion, 2016
4. Beighley Lynn: Head First. SQL. Edycja polska., Wydanie I, Helion, 2011
5. Cay S. Horstmann: Java. Techniki zaawansowane., Wydanie X, Helion, 2017
6. Cay S. Horstmann, Gary Cornell: Java. Podstawy., Wydanie IX, Helion, 2013
7. Downey Allen B., Mayfield Chris: Myśl w języku Java! Nauka programowania., Wydanie I, Helion, 2017
8. Heffelfinger, David R. : Java EE 6. Tworzenie aplikacji w NetBeans 7, Wydanie , Helion, 2014
9. Horstmann Cay S.: Java. Podstawy., Wydanie X, Helion, 2016
10. Lis Marcin: Praktyczny kurs Java, Wydanie IV, Helion, 2015
11. Lis Marcin: Java. Leksykon kieszonkowy., Wydanie II, Helion, 2007
12. Martin Fowler: Patterns of Enterprise Application Architecture. Data Transfer Object., Wydanie I, Addison-Wesley, 2010
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. Schmuller Joseph: UML dla każdego., Wydanie I, Helion, 2003
16. Sierra Kathy, Bates Bert: Head First. Java. Edycja polska., Wydanie II, Helion, 2011
17: URL, W3Schools Online Web Tutorials, (dostęp: wrzesień 2019), https://www.w3schools.com/
18: URL, Date Range Picker — JavaScript Date & Time Picker Library, (dostęp: październik 2019), https://www.daterangepicker.com/

5 Spis załączników

Tabele

Wykaz rysunków

Wykaz listingów