



Politechnika Łódzka

Instytut Informatyki

RAPORT Z PROJEKTU KOŃCOWEGO

STUDIÓW PODYPLOMOWYCH

NOWOCZESNE APLIKACJE BIZNESOWE JAVA EE

**System informatyczny obsługujący sprzedaż biletów
na koncerty**

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej

Opiekun: dr inż. Mateusz Smoliński

Słuchacz: inż. Paulina Kubiak

Łódź, 06.12.2019



Instytut Informatyki

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

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

Spis treści

1	Cel i zakres projektu.....	3
2	Założenia projektu.....	4
2.1	Wersje zastosowanych technologii i narzędzi.....	4
2.2	Wymagania funkcjonalne.....	4
2.2.1	Diagramy przypadków użycia.....	4
2.2.2	Tabela krzyżowa ról i przypadków użycia.....	6
2.2.3	Opis przypadków użycia.....	7
2.2.4	Realizacja przykładowego CRUD.....	15
2.3	Wymagania niefunkcjonalne.....	26
3	Realizacja projektu.....	27
3.1	Warstwa składowania danych.....	27
3.1.1	Model relacyjnej bazy danych.....	27
3.1.2	Konfiguracja zasobów relacyjnej bazy danych.....	28
3.1.3	Klasy encyjne. Mapowanie obiektowo – relacyjne ORM.....	28
3.2	Warstwa logiki biznesowej.....	30
3.2.1	Komponenty EJB.....	30
3.2.2	Transakcyjność, blokady optymistyczne.....	33
3.2.3	Bezpieczeństwo.....	33
3.2.4	Obsługa błędów.....	34
3.3	Warstwa widoku.....	36
3.3.1	Wzorzec projektowy DTO.....	36
3.3.2	Ujednolicony wygląd stron.....	37
3.3.3	Internacjonalizacja.....	37
3.3.4	Uwierzytelnienie, autoryzacja.....	38
3.3.5	Walidacja danych.....	40
3.3.6	Obsługa błędów.....	41
3.4	Instrukcja wdrożenia.....	42
3.5	Podsumowanie.....	46
4	Źródła.....	48
5	Spis załączników.....	49

1 Cel i zakres projektu

Celem projektu jest zbudowanie trójwarstwowego systemu informatycznego umożliwiającego zakup biletów na koncerty. System będzie składał się z aplikacji internetowej oraz relacyjnej bazy danych. Aplikacja zostanie wykonana w technologii Java Enterprise Edition, osadzona w lokalnym serwerze aplikacyjnym Payara a relacyjna baza danych zostanie zlokalizowana w lokalnym systemie zarządzania bazami danych JavaDB. Serwer aplikacyjny Payara jak i system zarządzania bazami danych będą zlokalizowane w tym samym systemie operacyjnym. Dostęp do aplikacji będzie możliwy poprzez interfejs użytkownika, który wymaga wykorzystania współczesnej przeglądarki internetowej. System informatyczny jest wielodostępny oraz obsługuje różne poziomy dostępu użytkowników.

W systemie wyróżniamy cztery poziomy dostępu:

- Gość - użytkownik niezalogowany,
- Klient – dokonuje zakupu,
- Pracownik – dodaje koncerty oraz definiuje dostępną pulę biletów,
- Administrator – zarządza kontami,

W systemie użytkownik musi posiadać indywidualne konto z przypisanym pojedynczym poziomem dostępu spośród wymienionych. Uwierzytelnienie w systemie bazuje na podaniu loginu i hasła przypisanych do kont użytkownika, wzorowe dane do uwierzytelnienia wraz z przypisanym do konta poziomem dostępu przechowywane są w relacyjnej bazie danych. Wzorce haseł w bazie są przechowywane jako skróty wyliczone algorytmem funkcji jednokierunkowej SHA256.

Zakres projektu obejmuje:

- Opracowanie założeń i wymagań biznesowych,
- Dobór technologii,
- Zaprojektowanie struktur i implementacja dla bazy relacyjnej,
- Projekt oraz implementacja aplikacji,
- Stworzenie instrukcji wdrożenia,
- Uruchomienie systemu,
- Przygotowanie danych inicjujących dla bazy danych,
- Podsumowanie,

2 Założenia projektu

System ma zapewnić możliwość rejestracji koncertów przez pracownika oraz możliwość zakupu biletu z puli przypisanej do wybranego przez klienta koncertu. Jeden klient może kupić dowolną liczbę biletów z dostępnych w puli oraz dokonać jego zwrotu. Sprzedaż biletów kończy się na 24 godziny przed rozpoczęciem wydarzenia.

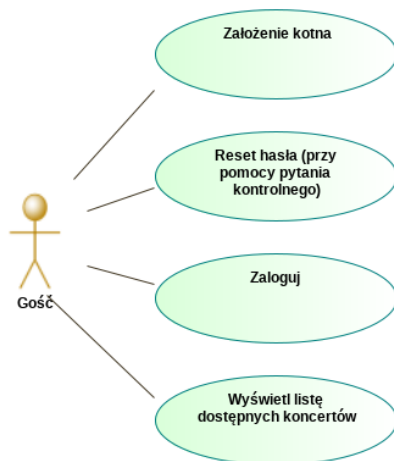
2.1 Wersje zastosowanych technologii i narzędzi

- Środowisko programistyczne NetBeans IDE w wersji 8.2;
- Język programowania Java w wersji 8;
- JDK (Java Development Kit) w wersji 1.8;
- Maven w wersji 3.6.0 – budowanie i zarządzanie zależnościami w projekcie;
- Serwer aplikacyjny Payara 5.184;
- Java Enterprise Edition w wersji 7.0 – platforma programistyczna umożliwiająca tworzenie aplikacji w architekturze kontener-komponent (tworzona logika biznesowa);
- Ziarna CDI (Context and Dependency Injection) w wersji 1.2 – warstwa pośrednicząca (kontroler) pomiędzy warstwą prezentacji a warstwą logiki biznesowej;
- Ziarna EJB (Enterprise JavaBeans) 3.2 – warstwa logiki biznesowej;
- Java Server Faces 2.2 - obsługa warstwy widoku;
- Java PrimeFaces 7.0
- Java DB (Derby) 10.13 – system zarządzania bazą danych;
- Java Persistence API 2.5.2 (mapowanie obiektowo-relacyjne);
- Modelio 3.7;

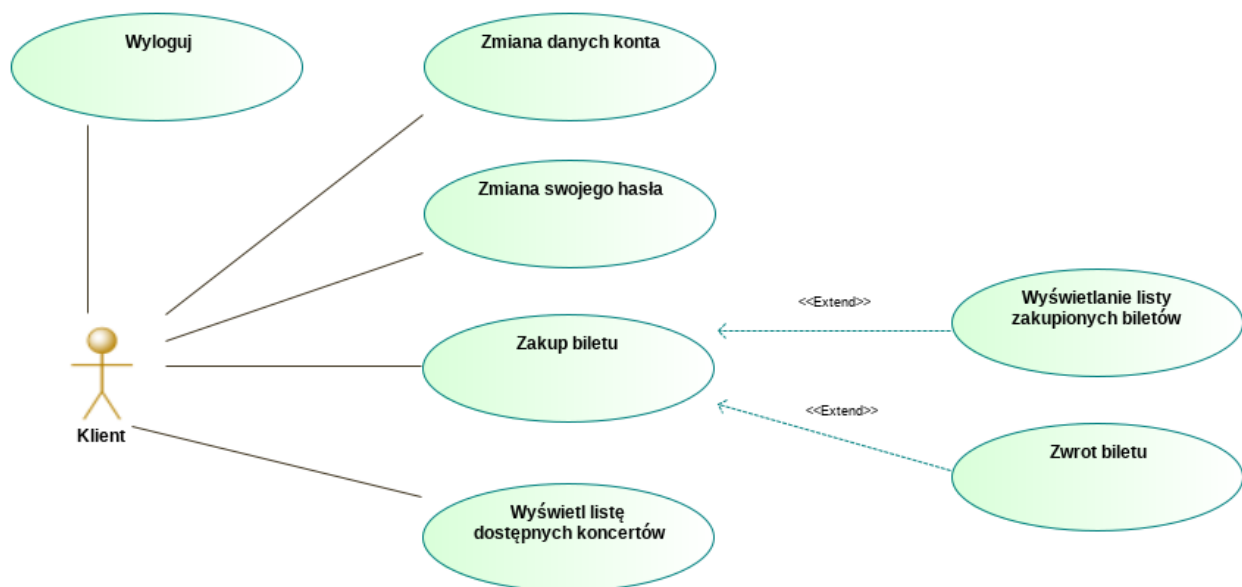
2.2 Wymagania funkcjonalne

2.2.1 Diagramy przypadków użycia

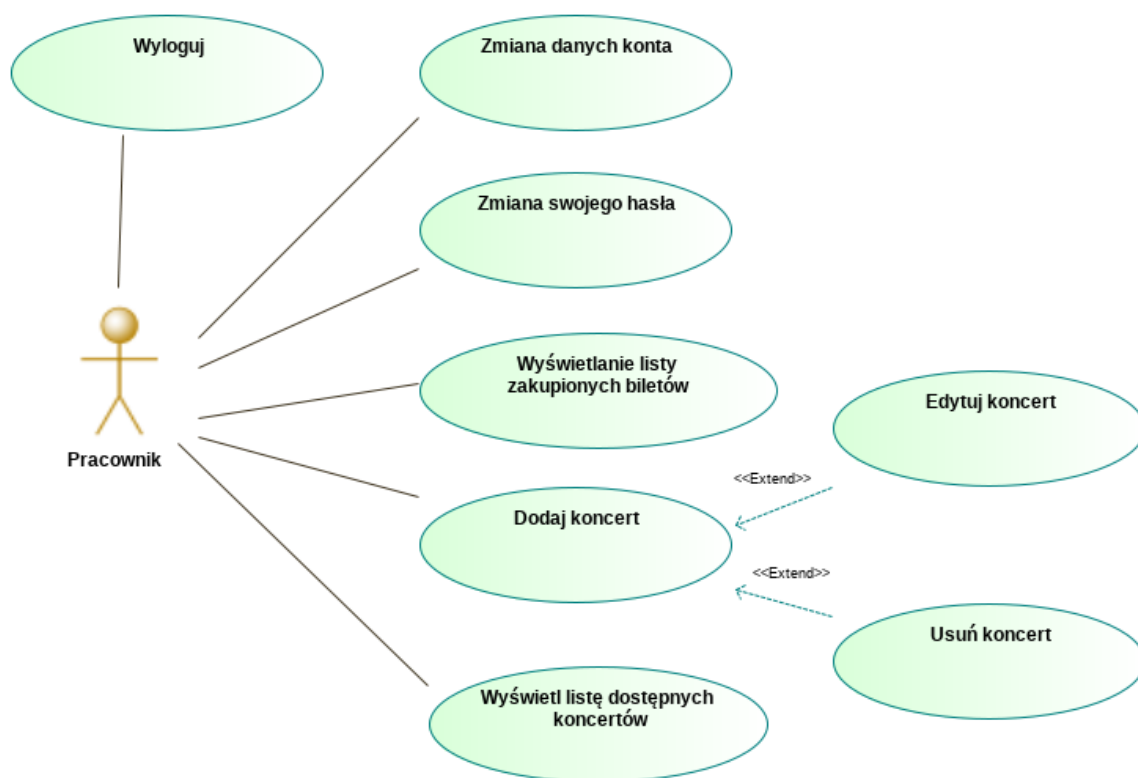
Wymagania funkcjonalne przedstawiono w postaci diagramów przypadków użycia. Na rysunku 1 poziom dostępu Gość, na rysunku 2 poziom dostępu Klient, na rysunku 3 poziom dostępu Pracownik, na rysunku 4 poziom dostępu Administrator.



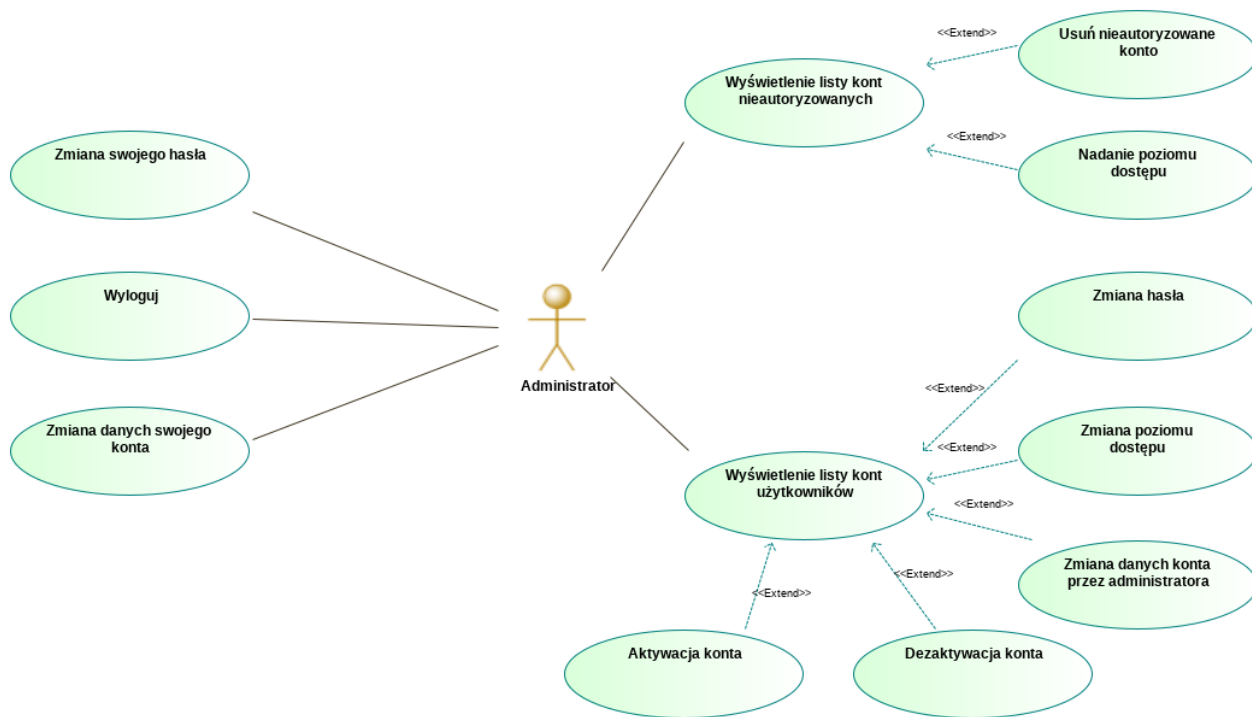
Rysunek 1: Przypadki użycia dla poziomu dostępu Gość



Rysunek 2: Przypadki użycia dla poziomu dostępu Klient



Rysunek 3: Przypadki użycia dla poziomu dostępu Pracownik



Rysunek 4: Przypadki użycia dla poziomu dostępu Administrator

2.2.2 Tabela krzyżowa ról i przypadków użycia

W tabeli 1 zostały przedstawione wszystkie przypadki użycia wraz z przypisanymi poziomami dostępu.

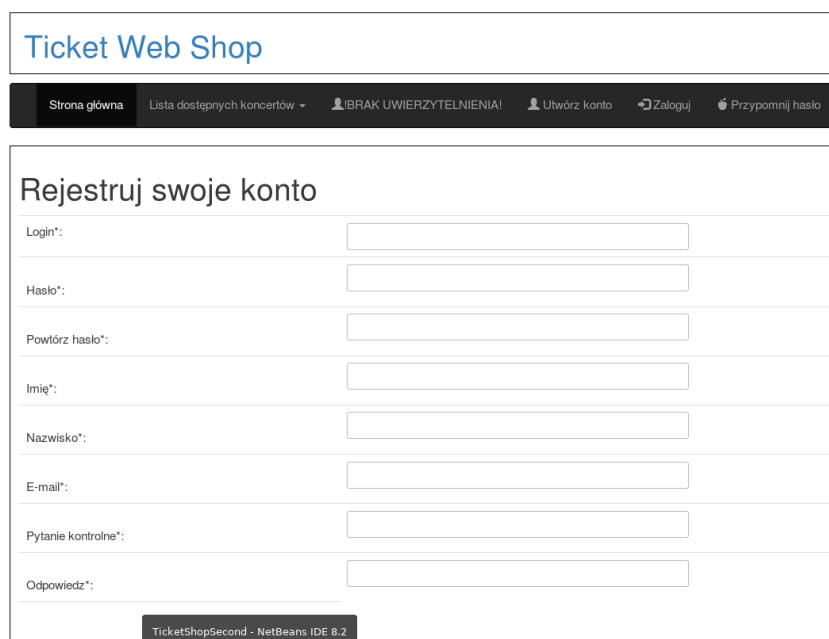
Tabela 1. Tabela krzyżowa ról i przypadków użycia

Lp.		Przypadki użycia	Gość	Klient	Pracownik	Administrator
1	konto	Rejestracja konta	x			
2	konto	Aktywacja konta				x
3	konto	Dezaktywacja konta				x
4	konto	Reset hasła (przy pomocy pytania kontrolnego)	x			
5	konto	Zaloguj	x			
6	konto	Wyloguj		x	x	x
7	konto	Zmiana danych swojego konta		x	x	x
8	konto	Zmiana swojego hasła		x	x	x
9	konto	Zmiana hasła				x
10	konto	Wyświetlenie listy kont nieautoryzowanych				x
11	konto	Wyświetlenie listy kont użytkowników				x
12	konto	Usun nieautoryzowane konto				x
13	konto	Zmiana poziomu dostępu				x
14	konto	Nadanie poziomu dostępu				x
15	konto	Zmiana danych konta przez administratora				x
16	bilet	Wyświetlanie listy wszystkich zakupionych biletów			x	
17	bilet	Wyświetlanie listy zakupionych przez klienta		x		
18	bilet	Zakup biletu		x		
19	bilet	Zwrot biletu		x		
20	koncert	Rejestracja koncertu			x	
21	koncert	Edytuj koncert			x	
22	koncert	Wyświetl listę dostępnych koncertów	x	x	x	
23	koncert	Usun koncert			x	

2.2.3 Opis przypadków użycia

W rozdziale znajdują się opisy wszystkich przypadków użycia:

- Rejestracja konta – użytkownik nieposiadający konta w systemie wypełnia formularz z wymaganymi polami (Rysunek 5). Następnie wprowadzone dane zostają zapisane w tabeli Konto w relacyjnej bazie danych z tymczasowym poziomem dostępu nowe konto (bez dostępu do jakichkolwiek funkcjonalności). Model bazy danych został zaprezentowany w rozdziale 3.1.1. Scenariusz błędu: podanie istniejącego w bazie loginu, brak wypełnienia wymaganego pola, wpisanie hasła zawierającego mniej niż 8 znaków (w tym jedna duża litera, jeden znak specjalny oraz cyfra), niezgodność w polach „Hasło” oraz „Powtórz hasło”.



The screenshot shows a web application titled "Ticket Web Shop". Below the title is a navigation bar with links: "Strona główna", "Lista dostępnych koncertów", "IBRAK UWIERZYTELNIENIA!", "Utwórz konto", "Zaloguj", and "Przypomnij hasło". The main content area is titled "Rejestruj swoje konto" and contains a registration form with the following fields: "Login*", "Hasło*", "Powtórz hasło*", "Imię*", "Nazwisko*", "E-mail*", "Pytanie kontrolne*", and "Odpowiedz*". At the bottom of the form, there is a small text box that says "TicketShopSecond - NetBeans IDE 8.2".

Rysunek 5: Formularz rejestracji nowego konta

- Aktywacja konta – dokonuje Administrator, prowadzi do ustawienia konta aktywnym, poprzez zmianę wartości pola aktywne na prawdziwe (ang. *true*) w tabeli Konto. Konto aktywne posiada dostęp do funkcjonalności.
- Dezaktywacja konta – dokonuje Administrator, prowadzi do ustawienia konta nieaktywnym, poprzez ustawienie wartości pola aktywne na nieprawdziwe (ang. *false*) w tabeli Konto. Konto nieaktywne nie ma dostępu do funkcjonalności.

Ticket Web Shop

[Strona główna](#)[admin](#)[Administrator](#)[Wyloguj](#)

Lista kont aktywnych

Login	Imię	Nazwisko	E-mail	Akcje
admin	Jan	Nowak	admin@shop.pl	administrator Zmiana poziomu dostępu Edycja konta Zmiana hasła Aktywuj konto Dezaktywuj konto
test	Paulina	Kubiak	paulina@tss.com	klient Zmiana poziomu dostępu Edycja konta Zmiana hasła Aktywuj konto Dezaktywuj konto
test2	Paulina	Kubiak	paulina@tss.com	klient Zmiana poziomu dostępu Edycja konta Zmiana hasła Aktywuj konto Dezaktywuj konto
pracownikJan	Jan	Nowak	pracownik@shop.pl	pracownik Zmiana poziomu dostępu Edycja konta Zmiana hasła Aktywuj konto Dezaktywuj konto
klient	Jann	Nowak	klient@shop.pl	klient Zmiana poziomu dostępu Edycja konta Zmiana hasła Aktywuj konto Dezaktywuj konto
test10	Paulina	Kubiak	paulina@tss.com	pracownik Zmiana poziomu dostępu Edycja konta Zmiana hasła Aktywuj konto Dezaktywuj konto
test3	Paulina	Kubiak	paulina@tss.com	pracownik Zmiana poziomu dostępu Edycja konta Zmiana hasła Aktywuj konto Dezaktywuj konto

[Powrót do strony głównej](#)

Rysunek 6: Lista kont z przypisanym poziomem dostępu, funkcjonalność aktywuj oraz dezaktywuj wybrane konto

- Reset hasła przy pomocy pytania kontrolnego – funkcjonalność przeznaczona dla użytkownika, który posiada zarejestrowane konto w systemie, lecz nie pamięta swojego hasła. W pierwszym etapie resetu hasła należy podać swój login, który jest sprawdzany w bazie. Po pozytywnej weryfikacji zostajemy przeniesieni na stronę drugiego etapu, w którym jesteśmy proszeni o podanie odpowiedzi na pytanie kontrolne (ustawiane podczas rejestracji konta). Następnie możemy ustalić nowe hasło, które zostanie zapisane w bazie danych. Formularz resetu hasła został przedstawiony na rysunku 7.

Scenariusz błędu: brak wypełnienia wymaganego pola, podanie nieistniejącego w bazie danych loginu, podanie błędnej odpowiedzi na pytanie kontrolne, wpisanie hasła zawierającego mniej niż 8 znaków (w tym jedna duża litera, jeden znak specjalny oraz cyfra), brak zgodności haseł w polach „Hasło” oraz „Powtórz hasło”.

Ticket Web Shop

[Strona główna](#) [Lista dostępnych koncertów](#) [BRAK UWIERZYTELNIENIA!](#) [Utwórz konto](#) [Zaloguj](#) [Przypomnij hasło](#)

Login*: test3

Resetuj hasło

Anuluj

[Strona główna](#) [Lista dostępnych koncertów](#) [BRAK UWIERZYTELNIENIA!](#) [Utwórz konto](#) [Zaloguj](#) [Przypomnij hasło](#)

Login*: test3

Pytanie kontrolne*: Imię matki?

Odpowiedz*:

Nowe hasło*:

Powtórz nowe hasło*:

Resetuj hasło

Anuluj

Rysunek 7: Zmiana swojego hasła przy pomocy pytania kontrolnego

- Zaloguj – Użytkownik proszony jest o podanie swojego unikalnego loginu oraz hasła, poprawność danych zostaje sprawdzona w bazie i wówczas zostaje otwarta sesja użytkownika. Formularz do logowania został przedstawiony na rysunku 8. Po zalogowaniu użytkownik ma dostęp do wybranych dla jego poziomu dostępu funkcjonalności. Scenariusz błędu: podanie loginu oraz hasła, które nie jest zapisane w bazie danych, podanie błędnego loginu i hasła znajdującego się w bazie danych, brak wypełnienia wymaganych pól.

Rysunek 8: Formularz logowanie do systemu

- Wyloguj – Dokonuje zamknięcia bieżącej sesji użytkownika. Użytkownik musi potwierdzić chęć wylogowania z systemu zaprezentowane na rysunku 9. Automatyczne wylogowanie z systemu nastąpi po upływie 10 minut. Po wylogowaniu użytkownik traci dostęp do funkcjonalności przypisanych do jego poziomu dostępu.

Rysunek 9: Wylogowanie z systemu

- Zmiana danych swojego konta – dane użytkownika zostają pobrane z tabeli Konto oraz poddane możliwości edytowania imienia, nazwiska oraz email. Login jest wartością unikalną i nie może ulec zmianie. Formularz edycji danych zaprezentowano na rysunku 10.

Rysunek 10: Formularz edycji danych swojego konta (na przykładzie konta Administrator)

- Zmiana swojego hasła – funkcjonalność dostępna dla zarejestrowanego oraz zalogowanego użytkownika. Po podaniu starego hasła, dwukrotnie nowego oraz zatwierdzeniu formularza przedstawionego na rysunku 11, nowe hasło zostaje zapisane w bazie w tabeli Konto. Scenariusz błędu: błędne podanie starego hasła, wpisanie hasła zawierającego mniej niż 8 znaków (w tym wielka litera, znak specjalny oraz cyfra), brak zgodność haseł w polach „Hasło” oraz „Powtórz hasło”, brak wypełnienia wymaganych pól.

Rysunek 11: Formularz zmiany hasła dla zalogowanego użytkownika

- Zmiana hasła – funkcjonalność przeznaczona dla poziomu dostępu „Administrator”. Administrator z listy kont wybiera konto, uzupełnia formularz przedstawiony na rysunku 12. Po zatwierdzeniu nowe hasło zostaje zapisane w tabeli Konto. Administrator musi przekazać użytkownikowi nowe hasło. Scenariusz błędu: brak wypełnienia wymaganego pola, wpisanie hasła zawierającego mniej niż 8 znaków (w tym wielka litera, znak specjalny oraz cyfra), brak zgodność haseł w polach „Hasło” oraz „Powtórz hasło”.

Rysunek 12: Formularz zmiany hasła przez administratora

- Wyświetlenie listy kont nieautoryzowanych – funkcjonalność przeznaczona dla poziomu dostępu Administrator. Pozwala na wyświetlenie listy nowych kont oraz wykonania na nich akcji widocznych na rysunku 14.
- Wyświetlenie listy kont użytkowników - funkcjonalność przeznaczona dla poziomu dostępu Administrator. Pozwala na wyświetlenie kont autoryzowanych, czyli z przypisanym poziomem dostępu oraz wykonania na nich akcji zaprezentowanych na rysunku 6.

- Usunięcie nieautoryzowanego konta – administrator ma możliwość usunięcia nowo zarejestrowanego konta przed wykonaniem na nim jakichkolwiek akcji. Po wybraniu akcji Usuń z listy widocznej na rysunku 14, administrator zostaje przekierowany na stronę z potwierdzeniem usunięcia konta rysunek 13. W przypadku usunięcia konta, dane usuwane są z bazy danych.

Rysunek 13: Usuwanie konta

- Nadanie poziomu dostępu - administrator z listy rozwijanej zaprezentowanej na rysunku 14 wybiera poziom dostępu dla konta. Przed przydzieleniem poziomu dostępu wiążącym się z jednoczesną autoryzacją, administrator potwierdza dane użytkownika. Po wybraniu dla konta poziomu dostępu konto trafia na listę kont użytkowników.

Rysunek 14: Ustawienie poziomu dostępu

- Zmiana poziomu dostępu – umożliwia administratorowi zmianę poziomu dostępu dla wybranego konta z listy, zaprezentowane na rysunku 6.
- Zmiana danych konta przez administratora – administrator posiada możliwość zmiany danych personalnych dla wybranego konta. Formularz do zmiany danych przedstawiono na rysunku 10.
- Wyświetlenie listy wszystkich biletów – prowadzi do wyświetlenia danych z tabeli Bilet z bazy danych. Na rysunku 15 został przedstawiony widok dla pracownika. Pracownik z listy rozwijanej wybiera wydarzenie i po zatwierdzeniu przyciskiem „Pokaż” zostają wyświetlone dane w tabeli.

Rysunek 17: Formularz zakupu biletu

- Zwrot biletu – klient posiada możliwość zwrotu dowolnej ilości zakupionych przez siebie biletów. W tym celu z listy zakupionych biletów należy wybrać przycisk „Zwróć bilet” widoczny na rysunku 16. Zwrot biletu oznacza usunięcie krotki z tabeli Bilet w bazie danych.
- Dodaj koncert – pracownik posiada możliwość rejestracji nowego koncertu, którego dane zostaną zapisane w tabeli Koncert w bazie danych. Formularz rejestracji koncertu został przedstawiony na rysunku 18. Scenariusz błędu: brak podania wymaganych pól, nieoprawny format daty.

Rysunek 18: Formularz rejestracji koncertu

- Edytuj koncert – pracownik z listy koncertów zaprezentowanej na rysunku 21 posiada możliwość wyboru koncertu do edycji. Formularz edycji został zaprezentowany na rysunku 19. Nowe dane zostają zapisane w tabeli Koncert w bazie danych. Scenariusz błędu: niepoprawny format daty.

Rysunek 19: Formularz edycji koncertu

- Wyświetl listę dostępnych koncertów – Rysunek 20 - prowadzi do pokazania użytkownikowi danych z tabeli Koncert.

Strona główna				Klient				Wyloguj			
---------------	--	--	--	--------	--	--	--	---------	--	--	--

Lista dostępnych koncertów									
Wykonawca	Data	Miejsce	Cena	Akcje					
Kult	Sun Nov 10 20:00:00 CET 2019	Dekompresja	100	Kup bilet					
Bajm	Tue Dec 10 20:00:00 CET 2019	Wytwórnia	50	Kup bilet					
Sting	Tue Dec 10 20:00:30 CET 2019	Atlas Arena	150	Kup bilet					
test2	Tue Dec 24 20:00:00 CET 2019	Wytwórnia	10	Kup bilet					
test5	Tue Dec 24 20:00:00 CET 2019	Wytwórnia	10	Kup bilet					
test6	Tue Dec 24 20:00:00 CET 2019	Wytwórnia	10	Kup bilet					
test7	Tue Dec 24 20:00:00 CET 2019	Wytwórnia	10	Kup bilet					
test7	Tue Dec 24 20:00:00 CET 2019	Wytwórnia	10	Kup bilet					
The Adicts	Tue Dec 24 20:00:00 CET 2019	Wytwórnia	100	Kup bilet					
The Adicts	Tue Dec 24 20:00:00 CET 2019	Dekompresja	10	Kup bilet					
test10	Tue Dec 24 20:00:00 CET 2019	Wytwórnia	10	Kup bilet					

[Powrót do strony głównej](#)

Braku możliwości wybrania przycisku "Kup bilet", oznacza brak dostępnych biletów na wydarzenie (wszystkie bilety zostały już wyprzedane).

Rysunek 20: Lista dostępnych koncertów (widok dla klienta)

- Usuń koncert – z listy dostępnych koncertów, zaprezentowanej na rysunku 21 pracownik posiada możliwość wybrania akcji „Usuń koncert”. Po jej wybraniu zostajemy przekierowani na stronę z potwierdzeniem usunięcia koncertu. Zapisanie prowadzi do usunięcia krotki w tabeli Koncert w bazie danych. Usunięcie koncertu, na który został sprzedany minimum jeden bilet staje się nie możliwe.

Strona główna

pracownik

Pracownik

Wyloguj

Lista dostępnych koncertów

Wykonawca	Data	Miejsce	Pula biletów	Cena	Akcje
Kult	Sun Nov 10 20:00:00 CET 2019	Dekompresja	100	100	<div>Usuń koncert</div> <div>Edytuj</div>
Bajm	Tue Dec 10 20:00:00 CET 2019	Wytwórnia	150	50	<div>Usuń koncert</div> <div>Edytuj</div>
test	Tue Dec 24 20:00:00 CET 2019	test	1	1	<div>Usuń koncert</div> <div>Edytuj</div>
Sting	Tue Dec 10 20:00:30 CET 2019	Atlas Arena	1000	150	<div>Usuń koncert</div> <div>Edytuj</div>
test2	Tue Dec 24 20:00:00 CET 2019	Wytwórnia	100	10	<div>Usuń koncert</div> <div>Edytuj</div>
Kult	Tue Dec 24 20:00:00 CET 2019	Atlas Arena	1000	100	<div>Usuń koncert</div> <div>Edytuj</div>
test4	Tue Dec 24 20:00:00 CET 2019	Wytwórnia	1	100	<div>Usuń koncert</div> <div>Edytuj</div>

Powrót do strony głównej

Czy na pewno chcesz usunąć koncert?

Wykonawca: test4

Data [RRRR-MM-DD HH:mm]: Tue Dec 24 20:00:00 CET 2019

Miejsce: Wytwórnia

Pula biletów: 1

Cena biletu [zł]: 100

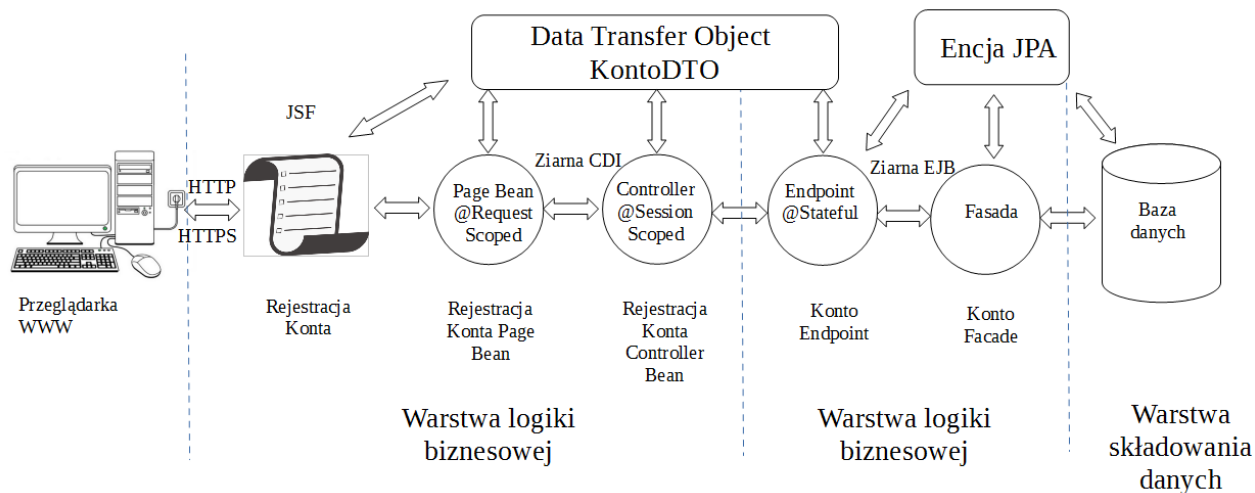
Usuń koncert

Powrót do poprzedniej strony

Rysunek 21: Usuń koncert

2.2.4 Realizacja przykładowego CRUD

W rozdziale zostanie opisany sposób realizacji przykładowego pełnego CRUD (z ang. *create, read, update, delete*) dla przypadków użycia związanych z kontem. Na rysunku 22 został przedstawiony diagram komponentów dla realizacji przykładowego CRUDa.



Rysunek 22: Diagram komponentów – utwórz konto

Create – Przypadek zostanie omówiony na przykładzie tworzenia rejestrowania konta użytkownika. Utwórz konto, czyli dodanie nowych danych do tabeli KONT0. Listing 1 przedstawia stronę JSF `rejestracjaKonta.xhtml`, która odpowiada za udostępnienie formularza do rejestracji konta. Widok z przeglądarki internetowej został zaprezentowany na rysunku 5. Atrybut *value* wskazuje na ziarno CDI odpowiedzialne za utworzenie konta.

```
<h:form id="rejestracjaKonta" styleClass="center_content">
<h1> ${msg['strona.rejestracja.konta.formularz.akcja.rejestruj']} </h1>
    <h:panelGrid columns="2" styleClass="table">
#{msg['strona.rejestracja.konto.label.login']}*:
<h:inputText id="login" value="${rejestracjaKontaPageBean.kontoDTO.login}"
required="true" size="50" maxLength="50"
requiredMessage="#{msg['strona.rejestracja.konto.formularz.login']}">
</h:inputText>
#{msg['strona.rejestracja.konto.label.haslo']}*:
<h:inputSecret id="haslo" value="${rejestracjaKontaPageBean.kontoDTO.haslo}"
required="true" size="50" maxLength="64"
validatorMessage="#{msg['strona.rejestracja.konto.formularz.walidacja.haslo']}
"requiredMessage="#{msg['strona.rejestracja.konto.formularz.haslo']}">
<f:validateRegex pattern="( (?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[@#!$%]).
{6,20})" /> </h:inputSecret>
#{msg['strona.rejestracja.konto.label.powtorzhaslo']}*:
<h:inputSecret id="hasloRepeat" value="$
{rejestracjaKontaPageBean.potworzHaslo}" required="true" size="50"
maxLength="64"
requiredMessage="#{msg['strona.rejestracja.konto.formularz.powtorz.haslo']}">
</h:inputSecret>
#{msg['strona.rejestracja.konto.form.label.imie']}*:
<h:inputText id="imie" value="${rejestracjaKontaPageBean.kontoDTO.imie}"
required="true" size="50" maxLength="50"
requiredMessage="#{msg['strona.rejestracja.konto.formularz.imie']}"/>
#{msg['strona.rejestracja.konto.form.label.nazwisko']}*:
<h:inputText id="nazwisko" value="$
```

```

{rejestracjaKontaPageBean.kontoDT0.nazwisko}" required="true" size="50"
maxlength="80"
requiredMessage="#{msg['strona.rejestracja.konto.formularz.nazwisko']}" />
#{msg['strona.rejestracja.konto.form.label.email']}*:
<h:inputText id="email" value="${rejestracjaKontaPageBean.kontoDT0.email}"
required="true" size="50" maxlength="150"
validatorMessage="#{msg['strona.rejestracja.konto.formularz.walidacja.email']}"
" requiredMessage="#{msg['strona.rejestracja.konto.formularz.email']}">
<f:validateRegex pattern="^[_a-z0-9-]+(\\.[_a-z0-9-]+)*@[a-z0-9-]+(\\.[a-z0-9-]
+)*\\.[a-z]{2,4}" /> </h:inputText>
#{msg['strona.rejestracja.konto.label.pytanie']}*:
<h:inputText id="pytanie" value="${rejestracjaKontaPageBean.kontoDT0.pytanie}"
required="true" size="50" maxlength="20"
requiredMessage="#{msg['strona.rejestracja.konto.formularz.pytanie']}" />
#{msg['strona.rejestracja.konto.label.odpowiedz']}*:
<h:inputText id="odpowiedz" value="$
{rejestracjaKontaPageBean.kontoDT0.odpowiedz}" required="true" size="50"
maxlength="150"
requiredMessage="#{msg['strona.rejestracja.konto.formularz.odpowiedz']}" />
</h:panelGrid>
<h:commandButton
value="#{msg['strona.rejestracja.konto.formularz.akcja.rejestruj']}" action="$
{rejestracjaKontaPageBean.rejestracjaKontaAction()}" />
<h:commandButton value="#{msg['strona.form.akcja.anuluj']}" action="main"
immediate="true" />
</h:form>

```

Listing 1: Fragment kodu źródłowego strony JSF rejestracjaKonta.xhtml

Listing 2 prezentuje ziarno CDI `RejestracjaKontaPageBean`. Metoda `rejestracjaKontaAction()` wywołuje metodę klasy `RejestracjaKontaControllerBean` zaprezentowanej na listingu 3. Atrybut `action` wskazuje na metodę wykonującą akcję. Adnotacja `@RequestScoped` nadaje ziarnu CDI zasięg żądania. Zasięg żądania jest ściśle powiązany z cyklem obsługi żądania i odpowiedzi HTTP [1]. Oznacza to, że rozpoczynając obsługę żądania kontener tworzy instancję klasy związanej z tym żądaniem. Po zakończeniu obsługi żądania instancja jest niszczonej przez kontener.

```

@Named(value = "rejestracjaKontaPageBean")
@RequestScoped
public class RejestracjaKontaPageBean implements Serializable {

    @Inject
    private RejestracjaKontaControllerBean rejestracjaKontaControllerBean;

    private KontoDT0 kontoDT0;
    private String potworzHaslo;
    (...)
    @PostConstruct
    public void init() {
        kontoDT0 = new KontoDT0();
    }

    public String rejestracjaKontaAction() {
        if (potworzHaslo != null && potworzHaslo.equals(kontoDT0.getHaslo()))
    {
        try {

```



```

        rejestracjaKontaControllerBean.rejestrujKonto(kontoDTO);
    } catch (AppBaseException ex) {

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

        ContextUtils.emitI18NMessage(null, ex.getMessage());
        return null;
    }
    } else {
        ContextUtils.emitI18NMessage("RegisterForm:passwordRepeat",
"konto.utworz.blad");
    }
    return "main";
}
}
}

```

Listing 2: Ziarno CDI RejestracjaKontaPageBean

Metoda rejestrujKonto klasy ControllerBean zaprezentowanej na Listing 3 ma za zadanie weryfikować, czy nie jest wykonywana ta sama czynność kilkakrotnie, poprzez sprawdzenie czy ten sam obiekt DTO nie jest po raz kolejny przekazywany. Ma to na celu uniknięcia ponownego zapisu tych samych danych w bazie. Adnotacja @SessionScoped nadaje ziarnu CDI zasięg sesji. Zasięg sesji obejmuje wiele cykli obsługi żądania i odpowiedzi HTTP (teoretycznie ich liczba nie jest niczym ograniczona) [1]. Oznacza to, że kontekst sesji jest niszczone dopiero po upływie czasu trwania sesji bądź po wylogowaniu się użytkownika z aplikacji.

```

@Named(value = "rejestracjaKontaController")
@SessionScoped
public class RejestracjaKontaControllerBean implements Serializable {
    @EJB
    private KontoEndpoint kontoEndpoint;
    (...)
    public void rejestrujKonto(final KontoDTO kontoDTO) throws AppBaseException {
        final int UNIQ_METHOD_ID = kontoDTO.hashCode() + 1;
        if (lastActionMethod != UNIQ_METHOD_ID) {
            int endpointCallCounter =
kontoEndpoint.NB_ATEMPTS_FOR_METHOD_INVOCATION;
            do {
                kontoEndpoint.rejestracjaKonta(kontoDTO);
                endpointCallCounter--;
            } while (kontoEndpoint.isLastTransactionRollback() &&
endpointCallCounter > 0);
            if (endpointCallCounter == 0) {
                throw
AppBaseException.createExceptionForRepeatedTransactionRollback();
            }
            ContextUtils.emitI18NMessage("RegisterForm:operationSuccess",
"error.sukces");
        } else {
            ContextUtils.emitI18NMessage("RegisterForm:repeatedAction",
"error.powtorz.akcje");
        }
        lastActionMethod = UNIQ_METHOD_ID;
    }
}

```

Listing 3: Ziarno CDI RejestracjaKontaControllerBean

Listing 4 przedstawia metodę rejestracjaKonta z klasy KontoEndpoint. Tworzony zostaje w niej nowy obiekt DTO.

```
@Stateful
@Transactional(TransactionalAttributeType.REQUIRES_NEW)
@Interceptors(LoggingInterceptor.class)
@RolesAllowed("Administrator")
public class KontoEndpoint extends AbstractEndpoint implements
SessionSynchronization {

    @Inject
    private KontoFacade kontoFacade;
    (...)
    @PermitAll
    public void rejestracjaKonta(KontoDTO kontoDTO) throws AppBaseException {
        Konto konto = new NoweKonto();
        konto.setImie(kontoDTO.getImie());
        konto.setNazwisko(kontoDTO.getNazwisko());
        konto.setEmail(kontoDTO.getEmail());
        konto.setLogin(kontoDTO.getLogin());
        konto.setHaslo(kontoDTO.getHaslo());
        konto.setPytaniekontrolne(kontoDTO.getPytanie());
        konto.setOdpowiedzkontrolna(kontoDTO.getOdpowiedz());
        konto.setAktywne(false);
        konto.setAutoryzowane(false);
        kontoFacade.create(konto);
    }
}
```

Listing 4: Fragment kodu źródłowego klasy KontoEndpoint

Listing 5 przedstawia fragment klasy AbstractFacade, która jest klasą nadrzędną wobec pozostałych klas Fasad. Na metodzie create wykonywana jest metoda persist, która odpowiada za utworzenie nowej krotki w tabeli KONTA w bazie danych.

```
public abstract class AbstractFacade<T> {

    private Class<T> entityClass;
    (...)
    public void create(T entity) throws AppBaseException {
        try {
            getEntityManager().persist(entity);
            getEntityManager().flush();
        } catch (DatabaseException e) {
            if (e.getCause() instanceof SQLNonTransientConnectionException) {
                throw
AppBaseException.createExceptionDatabaseConnectionProblem(e);
            } else {
                throw AppBaseException.createExceptionDatabaseQueryProblem(e);
            }
        } catch (Exception ex) {

            throw AppBaseException.createCreateObject(ex);
        }
    }
}
```

Listing 5: Metoda create klasy AbstractFacade

Read – Przypadek odczytu danych zostanie omówiony na przykładzie wyświetlenia listy kont nieautoryzowanych (świeżo założonych, nieaktywnych z automatycznie przydzielonym poziomem dostępu nowe konto) przez Administratora. Polega na odczytaniu bądź wyświetleniu zapisanych informacji z tabeli KONTA w bazie danych. Listing 6 przedstawia stronę JSF `listaRejestracjiKonta.xhtml`, która odpowiada za poprawną formę wyświetlenia danych w postaci tabeli. Widok z przeglądarki internetowej został zaprezentowany na rysunku 14. Atrybut `value` wskazuje na ziarno CDI odpowiedzialnego za utworzenie konta.

```
<h:form>
<h1> ${msg['strona.lista.nowych.kont.tytul']} </h1>
<h:dataTable var="row" value="${listaKontPageBean.dataModelAccounts}">
<h:column id="login"> <f:facet name="header">
{msg['strona.rejestracja.konto.formularz.label.login']}</f:facet>
<h:outputText value="#{row.login}" /> </h:column>
<h:column id="imie"> <f:facet name="header">${
{msg['strona.rejestracja.konto.formularz.label.imię']}</f:facet>
<h:outputText value="#{row.imie}" /> </h:column>
<h:column id="nazwisko"> <f:facet name="header">${
{msg['strona.rejestracja.konto.formularz.label.nazwisko']}</f:facet>
<h:outputText value="#{row.nazwisko}" /> </h:column>
<h:column id="email"> <f:facet name="header">${
{msg['strona.rejestracja.konto.form.label.email']}</f:facet>
<h:outputText value="#{row.email}" /> </h:column>
<h:column id="akcja"> <f:facet name="header">${
{msg['strona.lista.label.akcje']}</f:facet>
<h:selectOneMenu value="#{row.poziomDostepu}" > <f:selectItem itemLabel="$
{msg['strona.lista.nowych.kont.wybierz.poziom.dostepu']}" />
<f:selectItems value="${listaKontPageBean.listaPoziomowDostepu}"
var="poziomDostepu" itemValue="#{poziomDostepu}"
itemLabel="#{poziomDostepu.accessLevelI18NValue}" /> </h:selectOneMenu>
<h:commandButton value="${
{msg['strona.lista.nowych.kont.ustaw.poziom.dostepu.akcja']}" action="$
{listaKontPageBean.zmienPoziomDostepuKontoAction(row)}"/>
<h:commandButton value="{msg['strona.lista.nowe.konto.akcja.usun']}"
action="${listaKontPageBean.usunWybraneKontoAction(row)}"/>
</h:column> </h:dataTable>
<h:commandLink value="${msg['strona.akcja.powrot.do.głownej']}"
action="stronaGłowna.xhtml"/>
</h:form>
```

Listing 6: Fragment kodu źródłowego strony JSF `listaRejestracjiKonta.xhtml`

Listing 7 prezentuje ziarno CDI `ListaKontPageBean` o zasięgu widoku. Metoda `initListaNowychKont` oznaczona adnotacją `@PostConstruct` odpowiada za wyświetlenie poprawnych danych. Adnotacja `@ViewScoped` nadaje ziarnu CDI zasięg widoku. Zasięg widoku istnieje tak długo, jak długo będziemy się poruszać w obrębie tego samego widoku JSF wyświetlanego w przeglądarce lub na jej karcie[1]. Oznacza to, że komponenty w tym zasięgu są aktywne, do momentu kiedy nie przejdziemy na inną stronę.

```
@Named(value = "listaKontPageBean")
@ViewScoped
public class ListaKontPageBean implements Serializable {

    @EJB
    private KontoEndpoint kontoEndpoint;
```

```

private List<PoziomDostepu> listaPoziomowDostepu;
private List<KontoDTO> listaKont;
(...)
@PostConstruct
public void initListaNowychKont() {
    try {
        listaKont = kontoEndpoint.listaNowychKont();
    } catch (AppBaseException ex) {

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

        ContextUtils.emitI18NMessage(null, ex.getMessage());
    }
    dataModelAccounts = new ListDataModel<>(listaKont);
    PoziomDostepu[] listAllAccessLevels = PoziomDostepu.values();
    for (PoziomDostepu poziomDostepu : listAllAccessLevels) {

poziomDostepu.setAccessLevelI18NValue(ContextUtils.getI18NMessage(poziomDostep
u.getAccessLevelKey()));
    }
    listaPoziomowDostepu = new
ArrayList<>(Arrays.asList(listAllAccessLevels));
    listaPoziomowDostepu.remove(PoziomDostepu.KONTO);
    listaPoziomowDostepu.remove(PoziomDostepu.NOWEKONTO);
}
}
}
}

```

Listing 7: Fragment kodu klasy ListaKontPageBean

Listing 8 przedstawia metodę `listaNowychKont` klasy `KontoEndpoint`. W pierwszej kolejności zostaje wywołana metoda `znajdzAktywneKonta` z klasy `KontoFacade` Listing 9 oraz zwraca obiekty typu DTO, które zostają wyświetlone w tabeli na stronie JSF.

```

public List<KontoDTO> listaNowychKont() throws AppBaseException {
    List<Konto> listaNowychKont = kontoFacade.znajdzAktywneKonto();
    List<KontoDTO> listaNowychZarejestrowanychKont = new ArrayList<>();
    for (Konto konto : listaNowychKont) {
        KontoDTO kontoDTO = new KontoDTO(
konto.getLogin(),
konto.getImie(),
konto.getNazwisko(),
konto.getEmail());
        listaNowychZarejestrowanychKont.add(kontoDTO);
    }
    return listaNowychZarejestrowanychKont;
}

```

Listing 8: Fragment kodu klasy KontoEndpoint

Metoda `znajdzAktywneKonto` pobiera obiekty encji z bazy danych za pomocą kwerendy `findByNoweKonto` Listing 10 zwracając listę obiektów.

```

public List<Konto> znajdzAktywneKonto() throws AppBaseException {
    TypedQuery<Konto> tq = em.createNamedQuery("Konto.findByNoweKonto",
Konto.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 9: Fragment kodu klasy KontoFacade

Kwerenda `Konto.findByNoweKonto` Listing 10 pobiera dane z tabeli `KONTO` wszystkich nieautoryzowanych kont użytkowników.

```

@NamedQuery(name = "Konto.findByNoweKonto", query = "SELECT k FROM NoweKonto
k")

```

Listing 10: Kwerenda wyszukująca nowe konta z klasy encyjnej Konto

Update – Przypadek edycji zostanie omówiony na przykładzie edycji konta (modyfikacja oraz aktualizacja istniejących danych w bazie). Użytkownik z przypisanym poziomem dostępu Administrator ma możliwość zmiany a następnie aktualizacji, następujących danych użytkownika:

- imię,
- nazwisko,
- e-mail;

Pole login jest podstawiony na podstawie wcześniej wybranego konta z listy użytkowników. Nie ma możliwości edycji loginu. Możliwość edycji konta dotyczy tylko i wyłącznie kont autoryzowanych, czyli takich, które mają przypisany poziom dostępu. Każdorazowa edycja danych powoduje inkrementację wartości liczbowej +1 w polu `Wersja` w tabeli `KONTO`.

Listing 11 przedstawia stronę JSF `edytujKonto.xhtml`, która odpowiada za wyświetlenie formularza edycji konta. Widok z przeglądarki internetowej został zaprezentowany na rysunku 10.

```

<h:form id="edytujKonto" styleClass="center_content">
#{msg['strona.edytuj.konto.label.login']}: <h:outputText id="login" value="$
{edycjaKontaPageBean.kontoDTO.login}" /><br />
#{msg['strona.edytuj.konto.label.imie']}: <h:inputText id="imie" value="$
{edycjaKontaPageBean.kontoDTO.imie}" size="50" /><br />
#{msg['strona.edytuj.konto.label.nazwisko']}: <h:inputText id="nazwisko"
value="$#{edycjaKontaPageBean.kontoDTO.nazwisko}" size="50" /><br />
#{msg['strona.edytuj.konto.label.email']}: <h:inputText id="email"
value="$#{edycjaKontaPageBean.kontoDTO.email}" size="50" />
<h:commandButton value="#{msg['strona.form.akcja.zapisz']}"
action="#{edycjaKontaPageBean.zapiszEdycjeKontaAction()}" />
<h:commandButton value="#{msg['strona.form.akcja.anuluj']}"
action="listaKontAktywnych" immediate="true"/>
</h:form>

```

Listing 11: Fragment kodu strony JSF edytujKonto.xhtml

Listing 12 prezentuje ziarno CDI `EdycjaKontaPageBean` o zasięgu żądania.

```
@Named(value = "edycjaKontaPageBean")
@RequestScoped
public class EdycjaKontaPageBean {

    @EJB
    private KontoEndpoint kontoEndpoint;
    @Inject
    private RejestracjaKontaControllerBean rejestracjaKontaControllerBean;
    private KontoDTO kontoDTO;
    (...)
    @PostConstruct
    private void init() {
        kontoDTO = rejestracjaKontaControllerBean.getSelectedKontoDTO();
    }

    public String zapiszEdycjeKontaAction() {
        try {
            rejestracjaKontaControllerBean.edytujKonto(kontoDTO);
        } catch (AppBaseException ex) {

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

            ContextUtils.emitI18NMessage(null, ex.getMessage());
            return null;
        }

        return "listaKontAktywnych";
    }
}
```

Listing 12: Fragment kodu klasy `EdycjaKontaPageBean`

Listing 13 przedstawia metodę `edytujKonto` klasy `rejestracjaKontaControllerBean`, która ma za zadanie weryfikować, czy nie jest wykonywane ta sama czynność kilkakrotnie, poprzez sprawdzenie czy ten sam obiekt DTO nie jest po raz kolejny przekazywany. Ma to na celu uniknięcia ponownego zapisu tych samych danych w bazie. Został zastosowany mechanizm, który weryfikuje czy nie doszło do odwołania realizowanej transakcji aplikacyjnej, wówczas wykonanie metody zostaje ponowione. Sytuacja może się powtórzyć określoną liczbę razy po czym zgłaszany jest błąd.

```
public void edytujKonto(final KontoDTO kontoDTO) throws AppBaseException {
    final int UNIQ_METHOD_ID = kontoDTO.hashCode() + 4;
    if (lastActionMethod != UNIQ_METHOD_ID) {
        int endpointCallCounter =
kontoEndpoint.NB_ATEMPTS_FOR_METHOD_INVOCATION;
        do {
            kontoEndpoint.edytujKonto(kontoDTO);
            endpointCallCounter--;
        } while (kontoEndpoint.isLastTransactionRollback() &&
endpointCallCounter > 0);
        if (endpointCallCounter == 0) {
            throw
AppBaseException.createExceptionForRepeatedTransactionRollback();
        }
        ContextUtils.emitI18NMessage("RegisterForm:operationSuccess",
```

```

"error.sukces");
    } else {
        ContextUtils.emitI18NMessage("RegisterForm:repeatedAction",
"error.powtorz.akcje");
    }
    lastActionMethod = UNIQ_METHOD_ID;
}

```

Listing 13: Fragment kodu klasy RejestracjaKontaControllerBean

Listing 14 przedstawia metodę edytujKonto klasy KontoEndpoint. Porównuje login obecnego stanu z loginem zapisanym w bazie danych. Jeśli się zgadza to wówczas staje się możliwe przypisanie nowych wartości. Dodatkowo za pomocą setModyfikowanePrzez przypisuje w bazie danych login administratora, który dokonywał zmiany. Metoda edytujKonto wywołuje metodę edit klasy KontoFacade zaprezentowanej w listingu 15.

```

public void edytujKonto(KontoDTO kontoDTO) throws AppBaseException {
    if (kontoStan.getLogin().equals(kontoDTO.getLogin())) {
        kontoStan.setImie(kontoDTO.getImie());
        kontoStan.setNazwisko(kontoDTO.getNazwisko());
        kontoStan.setEmail(kontoDTO.getEmail());
        kontoStan.setModyfikowanePrzez(ladujBiezacegoAdmin());
        kontoFacade.edit(kontoStan);
    } else {
        throw KontoException.createExceptionWrongState(kontoStan);
    }
}

```

Listing 14: Fragment kodu klasy KontoEndpoint

```

@PermitAll
@Override
public void edit(Konto entity) throws AppBaseException {
    try {
        super.edit(entity);
    } catch (OptimisticLockException e) {
        throw AppBaseException.createExceptionOptimisticLock(e);
    } 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();
        throw AppBaseException.createExceptionDatabaseQueryProblem(cause);
    }
}

```

Listing 15: Fragment kodu klasy KontoFacade

Delete – Przypadek użycia prowadzący do usunięcia danych z bazy zostanie omówiony na przykładzie usunięcia nieautoryzowanego konta. Listing 17 przedstawia stronę JSF usunKonto.xhtml, która odpowiada za wyświetlenie strony potwierdzającej usunięcie konta. Widok z przeglądarki internetowej został zaprezentowany na rysunku 13.

```

<h:form>
<h:messages></h:messages>
#{msg['strona.usuniecie.potwierdzenie']} <br /><br />

```

```

#{msg['strona.edytuj.konto.label.login']}: <h:outputText id="login" value="$
{usunKontoPageBean.kontoDTO.login}" /><br />
#{msg['strona.edytuj.konto.label.imie']}: <h:outputText id="imie" value="$
{usunKontoPageBean.kontoDTO.imie}" /><br />
#{msg['strona.edytuj.konto.label.nazwisko']}: <h:outputText id="nazwisko"
value="{usunKontoPageBean.kontoDTO.nazwisko}" /><br />
<h:commandButton styleClass="button"
action="#{usunKontoPageBean.usunKontoAction()}"
value="#{msg['strona.lista.nowe.konto.akcja.usun']}" />
<h:commandButton value="#{msg['strona.akcja.powrot.do.glownej']}"
action="main" immediate="true"/>
</h:form>

```

Listing 16: Fragment kodu klasy usunKonto.xhtml

Listing 17 prezentuje fragment ziarna CDI UsunKontoPageBean o zasięgu żądania. Atrybut *action* w metodzie usunKontoAction wskazuje na metodę wykonującą akcję.

```

public String usunKontoAction() {
    try {
        rejestracjaKontaControllerBean.usunKonto(kontoDTO);
    } catch (AppBaseException ex) {
        Logger.getLogger(UsunKontoPageBean.class.getName()).log(Level.SEVERE, null,
ex);
        ContextUtils.emitI18NMessage(null, ex.getMessage());
        return null;
    }
    return "listaRejestracjiKonta";
}

```

Listing 17: Fragment kodu klasy UsunKontoPageBean

Listing 18 przedstawia metodę usunKonto klasy rejestracjaKontaControllerBean, która ma za zadanie weryfikować, czy nie jest wykonywane ta sama czynność kilkakrotnie, poprzez sprawdzenie czy ten sam obiekt DTO nie jest po raz kolejny przekazywany. Ma to na celu uniknięcia ponownego zapisu tych samych danych w bazie. Został zastosowany mechanizm, który weryfikuje czy nie doszło do odwołania realizowanej transakcji aplikacyjnej, wówczas wykonanie metody zostaje ponowione. Sytuacja może się powtórzyć określoną liczbę razy po czym zgłaszany jest błąd.

```

public void usunKonto(final KontoDTO kontoDTO) throws AppBaseException {
    final int UNIQ_METHOD_ID = kontoDTO.hashCode() + 2;
    if (lastActionMethod != UNIQ_METHOD_ID) {
        int endpointCallCounter =
kontoEndpoint.NB_ATEMPTS_FOR_METHOD_INVOCATION;
        do {
            kontoEndpoint.usunKonto(kontoDTO);
            endpointCallCounter--;
        } while (kontoEndpoint.isLastTransactionRollback() &&
endpointCallCounter > 0);
        if (endpointCallCounter == 0) {
            throw
AppBaseException.createExceptionForRepeatedTransactionRollback();
        }
        ContextUtils.emitI18NMessage("RegisterForm:operationSuccess",
"error.sukces");
    } else {
        ContextUtils.emitI18NMessage("RegisterForm:repeatedAction",

```



```

        "error.powtorz.akcje");
    }
    lastActionMethod = UNIQ_METHOD_ID;
}

```

Listing 18: Fragment kodu klasy RejestracjaKontaControllerBean

Listing 19 przedstawia metodę `usunKonto` klasy `KontoEndpoint`. Sprawdza czy wybrane konto w stanie jest aktywne, jeśli nie wówczas następuje wywołanie metody `remove` klasy `KontoFacade`.

```

public void usunKonto(KontoDTO kontoDTO) throws AppBaseException {
    Konto konto = kontoFacade.znajdzPoLoginie(kontoDTO.getLogin());
    if (konto.isAktywne() == false) {
        kontoFacade.remove(konto);
    } else {
        throw KontoException.createExceptionDeleted(konto);
    }
}

```

Listing 19: Fragment kodu klasy KontoEndpoint

Metoda `znajdzPoLoginie` pobiera obiekty encji z bazy danych za pomocą kwerendy `findByLogin` Listing 20 zwracając dany obiekt. Metoda `remove` odpowiada za usunięcie danej krotki w tabeli `KONTO` z bazy danych.

```

@PermitAll
public Konto znajdzPoLoginie(String login) throws AppBaseException {
    TypedQuery<Konto> tq = em.createNamedQuery("Konto.findByLogin",
Konto.class);
    tq.setParameter("lg", login);
    try {
        return tq.getSingleResult();
    } catch (NoResultException e) {
        throw KontoException.createExceptionNotExist(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);
        }
    }
}

@Override
public void remove(Konto entity) throws AppBaseException {
    try {
        super.remove(entity);
    } catch (DatabaseException e) {
        if (e.getCause() instanceof SQLNonTransientConnectionException) {
            throw
AppBaseException.createExceptionDatabaseConnectionProblem(e);
        } else {

```

```

throw AppBaseException.createExceptionDatabaseQueryProblem(e);
    }
}

```

Listing 20: Fragment kodu klasy KontoFacade.

Kwerenda `Konto.findByLogin` Listing 21 pobiera dane z tabeli KONT0 dla użytkownika o wybranym loginie.

```

@NamedQuery(name = "Konto.findByLogin", query = "SELECT k FROM Konto k WHERE
k.login = :lg")

```

Listing 21: Kwerenda wyszukująca konta po loginie z klasy encyjnej Konto

2.3 Wymagania niefunkcjonalne

System będzie spełniał następujące wymagania:

- System wymaga od użytkownika uwierzytelnienia się na wcześniej założone konto (utworzone i potwierdzone oraz aktywne). Bez uwierzytelnienia użytkownik systemu ma poziom dostępu gość.
- Autoryzacja do poszczególnych elementów widoku.
- System obsługuje wiele poziomów dostępu i jest wielodostępny.
- Poziom dostępu, który posiada użytkownik, ogranicza dostęp do funkcjonalności systemu.
- System obsługuje błędy.
- System zapewnia ochronę spójności danych poprzez wykorzystanie mechanizmu transakcji oraz blokad optymistycznych.
- Ujednolicony interfejs użytkownika do którego potrzebna jest współczesna przeglądarka internetowa WWW.
- System nie obsługuje polskich znaków diakrytycznych.
- Internacjonalizacja interfejsu użytkownika zostanie wykonana w języku polskim oraz angielskim.
- Dane biznesowe system przechowuje w relacyjnej bazie danych.
- System zgłasza zdarzenia dotyczące wykonywanych przez użytkowników akcji do systemowych dzienników zdarzeń, podobnie w dziennikach zdarzeń rejestrowane są błędy oraz granice transakcji aplikacyjnych.
- Wymagany jest bieżący czas ustawiony w systemie operacyjnym, gdzie działa serwer aplikacyjny.
- Dane przechowywane w relacyjnej bazie danych przy użyciu mapowania obiektowo-relacyjnego.
- Architektura zakładająca system trójwarstwowy.
- Transfer obiektów poprzez obiekty typu DTO.
- Klasy encyjne pomiędzy warstwą składowania danych a warstwą logiki biznesowej.
- System przechowuje w bazie danych informacje o dacie, godzinie oraz użytkownika, który dokonał ostatniej modyfikacji rekordu.

3 Realizacja projektu

3.1 Warstwa składowania danych

W projekcie do składowania danych została wykorzystana relacyjna baza danych. Do zarządzania relacyjną bazą danych została wykorzystana technologia Java DB.

3.1.1 Model relacyjnej bazy danych

Na rysunku 23 zostały przedstawione tabele odzwierciedlające model bazy danych. Tabela KONTA zawiera wszystkie informacje na temat kont użytkowników. Tabela KONCERT zawiera informację na temat koncertów. Natomiast tabela BILET przechowuje dane na temat zakupionych przez klientów biletów. Do tabeli BILET przypisane są klucze obce z tabel KONCERT oraz KLIENT. Każda z tabel znajduje odzwierciedlenie w klasach encyjnych zaprezentowanych na rysunku 24.

KONTO			
ID	BIGINT	PK	
KONTO	VARCHAR(30)	NN	
AKTYWNE	SMALLINT	NN	
AUTORYZOWANE	SMALLINT	NN	
DATA_MODYFIKACJI	DATE		
DATA_UTWORZENIA	TIMESTAMP	NN	
EMAIL	VARCHAR(255)	NN	
HASLO	VARCHAR(255)	NN	
IMIE	VARCHAR(255)	NN	
LOGIN	VARCHAR(255)	FK NN	
NAZWISKO	VARCHAR(255)	NN	
ODPOWIEDZ_KONTROLNA	VARCHAR(255)		
PYTANIE_KONTROLNE	VARCHAR(255)		
WERSJA	INTEGER		
MODYFIKOWANE_PRZEZ	BIGINT	FK NN	

KONCERT			
ID	BIGINT	PK	
CENA	INTEGER	NN	
DATA	TIMESTAMP	NN	
DATA_MODYFIKACJI	TIMESTAMP		
DATA_UTWORZENIA	TIMESTAMP	NN	
MIEJSCE	VARCHAR(255)	NN	
PULA_BILETÓW	INTEGER		
WERSJA	INTEGER	NN	
WYKONAWCA	VARCHAR(255)	NN	
MODYFIKOWANE_PRZEZ	BIGINT	FK	
WPROWADZ_KONCERT	BIGINT	FK	

BILET			
ID	BIGINT	PK	
DATA_MODYFIKACJI	DATE		
DATA_UTWORZENIA	TIMESTAMP	NN	
WERSJA	INTEGER		
KLIENT_ID	BIGINT	FK	
ID_KONCERTU	BIGINT	FK	

Wyjaśnienie skrótów:	
PK	Klucz główny (ang. Primary Key)
FK	Klucz obcy (ang. Foreign Key)
U	Wartość unikalna (ang. Unique)
NN	Wartość wymagana (ang. Not Null)

Rysunek 23: Model bazy danych

3.1.2 Konfiguracja zasobów relacyjnej bazy danych

Relacyjna baza danych w aplikacji obsługiwana jest poprzez JDBC (ang. *Java DataBase Connectivity*) oraz JPA (ang. *Java Persistence API*). Elementy składające się na konfigurację Java Persistence API dla aplikacji Java EE to: zasób serwera aplikacyjnego (pula połączeń JDBC, zasób JDBC), jednostka składowania (ang. *Persistence Unit*) oraz powołanie kontekstu trwałości (ang. *Persistence Context*). Plikiem konfiguracyjnym JPA wykorzystanym jest *persistence.xml* (ścieżka do pliku: *TicketShopSecond/src/main/resources/META-INF*). Na listingu 22 została przedstawiona konfiguracja pliku JPA dla aplikacji.

```
<persistence-unit name="TicketShopSecondPU" transaction-type="JTA">
  <jta-data-source>java:app/jdbc/TicketShopDS</jta-data-source>
  <exclude-unlisted-classes>false</exclude-unlisted-classes>
  <shared-cache-mode>NONE</shared-cache-mode>
  <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>
```

Listing 22: Zawartość pliku konfiguracyjnego *persistence.xml*

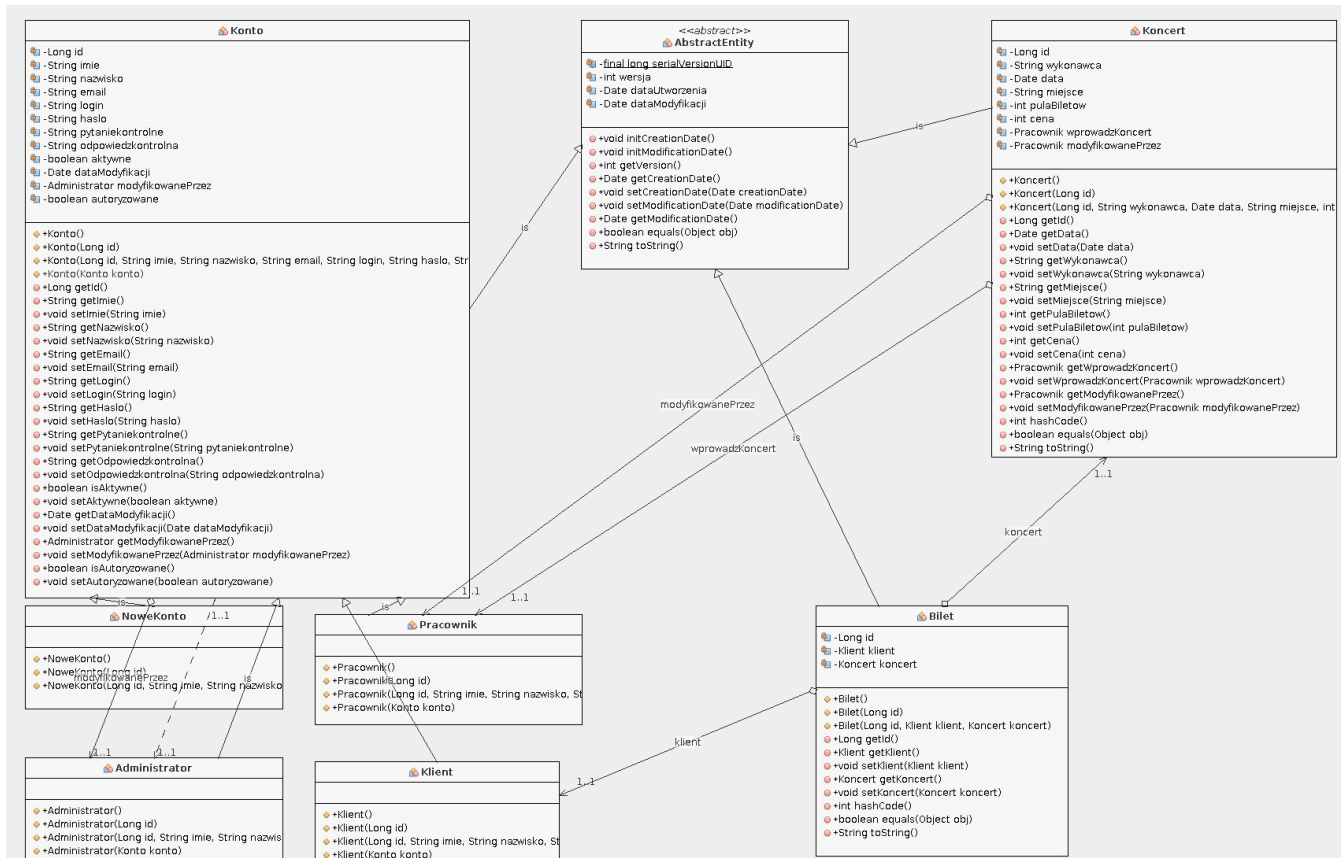
3.1.3 Klasy encyjne. Mapowanie obiektowo – relacyjne ORM.

Rysunek 24 przedstawia diagram klas encyjnych, strukturę bazy danych oraz powiązania między obiektami. Model danych uwzględnia encje wykonane w standardzie JPA (ang. *Java Persistence API*). Użycie JPA zapewnia m.in. prawidłowe odwzorowanie struktur baz danych za pomocą odpowiednich adnotacji znajdujących się w klasach encyjnych. Przykład użytych adnotacji został zaprezentowany na Listing 24.

Użycie w aplikacji klas encyjnych pozwoliło na zastosowanie modelu ORM (ang. *Object-Relational Mapping*). Mapowanie obiektowo – relacyjne to zautomatyzowane i przezroczyste utrwalenie w tabelach bazy danych SQL obiektów z aplikacji Javy z wykorzystaniem metadanych opisujących odwzorowanie pomiędzy klasami w aplikacji a schematem bazy danych SQL. Mechanizm ORM działa na zasadzie przekształcania (w odwracalny sposób) danych z jednej reprezentacji do innej [2].

W modelu tym, została zaimplementowana nadrzędna abstrakcyjna klasa bazowa, która rozszerza każdą z encji. Fragment kodu źródłowego tej klasy znajduje się na listingu 23, przedstawiając zaimplementowane pole wersji w celu obsługi mechanizmu blokad optymistycznych. Temat blokad optymistycznych rozstał szerzej opisany w rozdziale 3.2.2.

W klasie encyjnej KONT0 zaprezentowanej na listingu 24 zastosowano strategię dziedziczenia tabel. Dzięki takiemu rozwiązaniu tabela Konto rozszerza następujące klasy: Administrator, Klient, Nowe Konto, Pracownik.



Rysunek 24: Diagram klas encyjalnych

```

@MappedSuperclass
public abstract class AbstractEntity {

    private static final long serialVersionUID = 1L;

    @Version
    private int wersja;

```

Listing 23: Fragment kodu klasy bazowej AbstractEntity

```

@Entity
@Table(name = "KONTO")
@DiscriminatorColumn(name = "Konto", discriminatorType = DiscriminatorType.STRING)
@DiscriminatorValue(PoziomDostepu.KluczePoziomuDostepu.NOWEKONTO_KEY)

public class Konto extends AbstractEntity implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "KontoIdGen")
    @Column(name = "ID", nullable = false, unique = true)
    private Long id;

    @NotNull
    @Size(min = 1, max = 50)
    @Column(name = "IMIE", nullable = false)
    private String imie;

```

```

        @NotNull
        @Size(min = 1, max = 50)
        @Column(name = "LOGIN", nullable = false, unique = true, updatable =
false)
        private String login;

        @NotNull
        @Size(max = 64)
        @Column(name = "HASLO", nullable = false, updatable = true)
        private String haslo;

        @NotNull
        @Column(name = "AKTYWNE", nullable = false)
        private boolean aktywne;

        @NotNull
        @JoinColumn(name = "DATA_MODYFIKACJI", nullable = true)
        @OneToOne
        @Temporal(TemporalType.DATE)
        private Date dataModyfikacji;

        @NotNull
        @JoinColumn(name = "MODYFIKOWANEPRZEZ", nullable = true)
        @OneToOne
        private Administrator modyfikowanePrzez;

```

Listing 24: Fragment kodu klasy encyjnej Konto

3.2 Warstwa logiki biznesowej

Warstwa logiki biznesowej odpowiada za przetwarzanie danych zgodnie z założonym modelem biznesowym. W implementacji logiki biznesowej zostały wykorzystane ziarna EJB (ang. *Enterprise JavaBeans*). Komponenty EJB zostały szerzej zaprezentowane w rozdziale 3.2.1.

3.2.1 Komponenty EJB

Ziarna EJB wymagają kontenera EJB. Kontener ten zapewnia dodatkowe przydatne usługi, takie jak: zabezpieczenia, transakcje i współbieżność [3].

Komponenty EJB (ang. *Enterprise JavaBeans*) podzielić można na sesyjne komponenty stanowe (opatrzone adnotacją *@Stateful*) odpowiedzialne za transakcyjność, która szerzej została opisana w rozdziale 3.2.2 oraz bezstanowe fasady klas encyjnych (opatrzone adnotacją *@Stateless*).

Klasa *AbstractEndpoint* zaprezentowana na listingu 25 stanowi klasę bazową dla warstwy logiki biznesowej oraz jest przykładem stanowego ziarna EJB. Jest to klasa rozszerzająca pozostałe klasy *Endpoint*.

```

abstract public class AbstractEndpoint {

    @Resource
    SessionContext sctx;
    protected static final Logger LOGGER = Logger.getGlobal();

```

```

private String transactionId;

private boolean lastTransactionRollback;
public final int NB_ATEMPTS_FOR_METHOD_INVOCATION = 3;

public boolean isLastTransactionRollback() {
    return lastTransactionRollback;
}

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) {
    lastTransactionRollback = !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 25: Implementacja klasy bazowej AbstractEndpoint

AbstractFacade stanowi klasę bazową dla bezstanowych ziaren EJB implementujących fasady udostępniające podstawowe operacje na encjach JPA. Szczegóły implementacji zostały zaprezentowane na listingu 26.

```

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 {
        try {
            getEntityManager().persist(entity);
            getEntityManager().flush();
        } catch (DatabaseException e) {
            if (e.getCause() instanceof SQLNonTransientConnectionException) {
                throw
AppBaseException.createExceptionDatabaseConnectionProblem(e);
            } else {

```

```

        throw AppBaseException.createExceptionDatabaseQueryProblem(e);
    }
} catch (Exception ex) {
    throw AppBaseException.createCreateObject(ex);
}
}

```

Listing 26: Implementacja bazowej klasy AbstractFacade

W aplikacji został również zaimplementowany interceptor, którego zadaniem jest rejestracja zdarzeń zaistniałych podczas działania aplikacji. Zdarzenia zapisywane są jako komunikaty wywołanych metod biznesowych w dzienniku zdarzeń. Szczegóły implementacji interceptora znajdują się na listingu 27.

```

public class LoggingInterceptor {

    @Resource
    private SessionContext sessionContext;

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

```

Listing 27: Kod klasy LoggingInterceptor

3.2.2 Transakcyjność, blokady optymistyczne

Mechanizm transakcji w biznesowym systemie informatycznym gwarantuje spójność danych. Metody biznesowe komponentów EJB wykorzystują mechanizm zarządzania granicami transakcji przez kontener EJB, w strategii CMT (ang. *Container Management Transactions*). Został zaimplementowany interfejs `SessionSynchronization` w stanowych komponentach sesyjnych, dzięki któremu rejestrowane są granice transakcji w dziennikach zdarzeń. Dokonują się w nim zapisy stosownych komunikatów o rozpoczęciu oraz zakończeniu transakcji, a także wynik zakończenia transakcji (zatwierdzenie lub odwołanie). Poziom izolacji transakcji bazodanowych został określony na *read-committed* w pliku konfiguracyjnym przedstawionym w listingu 28.

```
<jdbc-connection-pool transaction-isolation-level="read-committed" (...) is-  
isolation-level-guaranteed="true" (...) </jdbc-connection-pool>
```

Listing 28: Fragment deskryptora wdrożenia glassfish-resources.xml

W systemie zaimplementowano mechanizm blokad optymistycznych mający na celu zapewnienie spójności danych. W przypadku, kiedy dwóch użytkowników w tym samym czasie będzie próbowało edytować ten sam obiekt, drugi który dokona próby zapisu spowoduje wystąpienie wyjątku typu *OptimisticLockException*. Obsługa tego wyjątku została zapewniona w klasach Fasad w metodach `edit` oraz `remove` Listing 15. Dzięki zastosowaniu mechanizmu blokad optymistycznych użytkownik aplikacji nie dokona operacji zmiany danych na danych nieaktualnych.

Do wykorzystania mechanizmu blokad optymistycznych niezbędnym była implementacja pola wersja. Zostało ono zaimplementowane w abstrakcyjnej klasie bazowej `AbstractEntity` w postaci adnotacji `@Version`, przedstawiono w listingu 23. Pole to zmienia swoją wartość przy każdej próbie edycji (inkrementacja wartości liczbowej).

3.2.3 Bezpieczeństwo

W aplikacji zastosowano model bezpieczeństwa kontroli dostępu opierający się na mapowaniu użytkowników i grup na role. Role mają odwzorowanie w poziomach dostępu. Mapowanie ról aplikacji zostało zapisane w deskrytorze wdrożenia serwera aplikacji szczegóły znajdują się w listingu 29. Natomiast dostęp ról do funkcjonalności systemu został określony w pliku konfiguracyjnym *web.xml*, który został zaprezentowany w listingu 35. Każda z ról posiada odrębny widok interfejsu graficznego użytkownika.

```
<glassfish-web-app error-url="">  
  <security-role-mapping>  
    <role-name>Administrator</role-name>  
    <group-name>administrator</group-name>  
  </security-role-mapping>  
  <security-role-mapping>  
    <role-name>Pracownik</role-name>  
    <group-name>pracownik</group-name>  
  </security-role-mapping>  
  <security-role-mapping>  
    <role-name>Klient</role-name>
```

```

        <group-name>klient</group-name>
    </security-role-mapping>
    <class-loader delegate="true"/>
    <jsp-config>
        <property name="keepgenerated" value="true">
            <description>Keep a copy of the generated servlet class' java
code.</description>
        </property>
    </jsp-config>
</glassfish-web-app>

```

Listing 29: Plik konfiguracyjny glassfish-web.xml

W warstwie logiki biznesowej został zastosowany model bezpieczeństwa deklaratywnego z wykorzystaniem adnotacji bezpieczeństwa `@RolesAllowed`. Przykład użycia został zaprezentowany w listingu 4, w którym zastosowano poziom adnotacji dla klasy `@RolesAllowed` („Administrator”) oraz metody `@PermitAll`. W tym przypadku oznacza, że pierwszeństwo ma adnotacja ustalona dla metody, następnie dla klasy.

3.2.4 Obsługa błędów

W aplikacji została zapewniona obsługa wyjątków aplikacyjnych. Zgłoszenie wyjątku w aplikacji powoduje automatyczne odwołanie bieżącej transakcji odpowiada za to atrybut `rollback=true` w adnotacji `@ApplicationException`. Zaprezentowana w listingu 30 klasa `AppBaseException` jest klasą bazową rozszerzającą pozostałe klasy obsługi błędów.

```

@ApplicationException(rollback = true)
public class AppBaseException extends Exception {

    static final public String KEY_OPTIMISTIC_LOCK =
"blad.blokada.optymistyczna.problem";
    static final public String KEY_DATABASE_QUERY_PROBLEM =
"blad.baza.danych.problem";
    static final public String KEY_DATABASE_CONNECTION_PROBLEM =
"blad.baza.danych.polaczenie.problem";
    static final public String KEY_ACTION_NOT_AUTHORIZED =
"blad.akcja.brak.autoryzacji.problem";

    private AbstractEntity pole;

    public AbstractEntity getPole() {
        return pole;
    }

    public AppBaseException(String message) {
        super(message);
    }

    public AppBaseException(AbstractEntity pole, String message) {
        super(message);
        this.pole = pole;
    }

    public AppBaseException(AbstractEntity pole, String komunikat, Throwable
powod) {
        super(komunikat, powod);
        this.pole = pole;
    }
}

```

```

        protected AppBaseException(String message, Throwable cause) {
            super(message, cause);
        }

        public static AppBaseException
        createExceptionDatabaseQueryProblem(Throwable e) {
            return new AppBaseException(KEY_DATABASE_QUERY_PROBLEM, e);
        }

        public static AppBaseException
        createExceptionDatabaseConnectionProblem(Throwable e) {
            return new AppBaseException(KEY_DATABASE_CONNECTION_PROBLEM, e);
        }

        public static AppBaseException
        createExceptionOptimisticLock(OptimisticLockException e) {
            return new AppBaseException(KEY_OPTIMISTIC_LOCK, e);
        }
    }
}

```

Listing 30: Fragment kodu źródłowego klasy bazowej AppBaseException

Na listingu 36 została zaprezentowana klasa obsługi wyjątków aplikacyjnych związanych z zakupem biletów. Przykład zgłoszenie wyjątku został zaprezentowany na listingu 20, natomiast obsługa wyjątku została szerzej omówiona w rozdziale 3.3.6.

```

public class BiletException extends AppBaseException {

    static final public String KEY_TICKET_SELL_OUT =
        "blad.bilet.wyprzedane.problem";
    static final public String KEY_TICKET_DAY =
        "blad.bilet.koniec.sprzedazy.problem";
    static final public String KEY_TICKET_NEGATIV_VALUE =
        "blad.bilet.wartosc.ujemna";
    static final public String KEY_TICKET_NOT_SOLD =
        "blad.bilet.brak.biletow";

    private Bilet bilet;

    public Bilet getBilet() {
        return bilet;
    }

    public BiletException(String message, Bilet bilet) {
        super(message);
        this.bilet = bilet;
    }

    public BiletException(Bilet bilet, String message, Throwable cause) {
        super(message, cause);
        this.bilet = bilet;
    }

    public BiletException(String message) {
        super(message);
    }

    public BiletException(String message, Throwable cause) {
        super(message, cause);
    }
}

```

```

static public BiletException createExceptionSellOut(Bilet bilet) {
    return new BiletException(KEY_TICKET_SELL_OUT, bilet);
}

static public BiletException createExceptionDay(Bilet bilet) {
    return new BiletException(KEY_TICKET_DAY, bilet);
}

static public BiletException createNegativValue(Bilet bilet) {
    return new BiletException(KEY_TICKET_NEGATIV_VALUE, bilet);
}

static public BiletException createNotSold(Bilet bilet) {
    return new BiletException(KEY_TICKET_NOT_SOLD, bilet);
}
}

```

Listing 31: Fragment kodu źródłowego klasy błędów aplikacyjnych BiletException

3.3 Warstwa widoku

Warstwa widoku odpowiada za przedstawienie w graficznym interfejsie użytkownika zapisanych oraz w odpowiedni sposób przetworzonych danych. Warstwa ta została zaimplementowana z wykorzystaniem standardu JSF (ang. *Java Server Faces*). JSF to cały framework do tworzenia aplikacji webowych, działający zgodnie z ideą wzorca projektowego MVC (ang. *Model-View-Controller*). Wobec tego JSF można określić jako zarządcę różnych współpracujących ze sobą technologii, któremu najbliższej do literki C – kontrolera, ma jednak też wiele wspólnego z V – czyli widokiem [4].

3.3.1 Wzorzec projektowy DTO

W aplikacji został wykorzystany model danych DTO (ang. *Data Transfer Object*) służący do transferowania obiektów pomiędzy warstwą prezentacji a warstwą logiki biznesowej. Dzięki takiemu rozwiązaniu dane mogą być od siebie odseparowane. Klasa DTO jest zwykłą klasą Javy, przykład klasy DTO przedstawiono na Listing 32.

```

public class BiletDTO {

    private long id;
    private String wykonawca;
    private Date data;
    private String miejsce;
    private int cena;
    private int iloscBiletow;
    private Date data_utworzenia;
    private KontoDTO klient;
    private KoncertDTO koncert;
    private int lp;
    private int iloscWolnych;

    public BiletDTO() {
    }

    public BiletDTO(long id, KontoDTO klient, KoncertDTO koncert) {
        this.id = id;
        this.klient = klient;
    }
}

```

```

        this.koncert = koncert;
    }

    public BiletDTO(long id, KontoDTO klient, KoncertDTO koncert, int lp) {
        this.id = id;
        this.klient = klient;
        this.koncert = koncert;
        this.lp = lp;
    }

    public BiletDTO(long id, KoncertDTO koncert) {
        this.id = id;
        this.koncert = koncert;
    }
    // publiczne metody get i set
}

```

Listing 32: Fragment kodu klasy BiletDTO

3.3.2 Ujednolicony wygląd stron

W aplikacji zostały wykorzystane technologie JSF (ang. *Java Server Faces*) oraz Bootstrap [5]. Dodatkowo został wykorzystany mechanizm Facelets Template odpowiadająca za ujednolicony interfejs graficzny użytkownika, który został zaprezentowany w rozdziale 2.2.3 przy omawianiu przypadków użycia. Każda ze stron WWW bazuje na jednolitym szablonie ustawiającym wspólny wygląd dla stron zbudowanego systemu informatycznego.

3.3.3 Internacjonalizacja

W interfejsie użytkownika została zaimplementowana internacjonalizacja zgodna ze standardem *i18n*. System zapewnia obsługę wielu języków, aktualnie zapewniono obsługę dwóch języków: polski oraz angielski. Język jest wybierany automatycznie na podstawie ustawień preferencji językowych w przeglądarce internetowej. W przypadku ustawienia preferencji języka, którego nie obsługuje system zostanie ustawiony język domyślny czyli język polski. Klucze oraz komunikaty wymagane do internacjonalizacji zostały zawarte w pliku *messages.properties*. Również niezbędna była dodatkowa konfiguracja w pliku *web.xml*

```

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

```

Listing 33: Konfiguracja w deskrytorze wdrożenia web.xml

Fragment kodu zaprezentowany na listingu 33 ma za zadanie wskazać, gdzie kontener ma szukać wartości dla klucza. Do każdej wersji językowej dodawany jest sufix określający język, np. *_en*. Pliki językowe są umieszczane w domyślnej lokalizacji *src/main/resources/i18n/*.

3.3.4 Uwierzytelnienie, autoryzacja

W systemie została wykorzystana metoda uwierzytelnienia poprzez formularz logowania. Strona `logowanie.xhtml` formularza logowania została zaprezentowana w listingu 34, natomiast widok został przedstawiony na rysunku 8. W zależności od przypisanej roli, zalogowany użytkownik ma dostęp do innej funkcjonalności aplikacji.

W zależności od poziomu dostępu, funkcjonalności przypisane do konta są różne. Mechanizm do obsługi uwierzytelnienia użytkownika został zapewniony przez serwer aplikacji Java EE, którego zadaniem jest weryfikacja tożsamości użytkownika. Na podstawie *TicketShopJDBCRealm* (źródło danych uwierzytelnienia, z którego ma korzystać mechanizm uwierzytelnienia) sprawdzana jest poprawność wprowadzonych w formularzu danych Listing 35. Konfiguracja *security realm* została zaprezentowana w rozdziale 3.4.

```
<form method="post" action="j_security_check">
<h1> ${msg['strona.zaloguj.podaj.login']} </h1>
<p>
#{msg['strona.rejestracja.konto.label.login']}*:
<input type="text" name="j_username" />
<h1> ${msg['strona.zaloguj.podaj.haslo']} </h1>
#{msg['strona.rejestracja.konto.label.haslo']}*
<input type="password" name="j_password" />
</p>
<p>
<input type="submit" value="${msg['strona.logowanie']}" action=
"stronaGlowna.xhtml" immediate="true"/>
</p>
</form>
```

Listing 34: Fragment kodu strony JSF `logowanie.xhtml`

```
<security-constraint>
  <display-name>Administrator</display-name>
  <web-resource-collection>
    <web-resource-name>Administrator</web-resource-name>
    <description/>
    <url-pattern>/faces/listaRejestracjiKonta.xhtml</url-pattern>
    <url-pattern>/faces/listaKontaAktywnych.xhtml</url-pattern>
    <url-pattern>/faces/zmienHaslo.xhtml</url-pattern>
    <url-pattern>/faces/edytujMojeKonto.xhtml</url-pattern>
    <url-pattern>/faces/zmienMojeHaslo.xhtml</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <description/>
    <role-name>Administrator</role-name>
  </auth-constraint>
</security-constraint>

<security-constraint>
  <display-name>Pracownik</display-name>
  <web-resource-collection>
    <web-resource-name>Pracownik</web-resource-name>
    <description/>
    <url-pattern>/faces/rejestracjaKoncertu.xhtml</url-pattern>
    <url-pattern>/faces/edytujMojeKonto.xhtml</url-pattern>
    <url-pattern>/faces/edytujKoncert.xhtml</url-pattern>
    <url-pattern>/faces/zmienMojeHaslo.xhtml</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <description/>
    <role-name>Pracownik</role-name>
  </auth-constraint>
</security-constraint>
```

```

        <url-pattern>/faces/rejestracjaKoncertu.xhtml</url-pattern>
        <url-pattern>/faces/usunKoncert.xhtml</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <description/>
        <role-name>Pracownik</role-name>
    </auth-constraint>
</security-constraint>

<security-constraint>
    <display-name>Klient</display-name>
    <web-resource-collection>
        <web-resource-name>Klient</web-resource-name>
        <description/>
        <url-pattern>/faces/kupBilet.xhtml</url-pattern>
        <url-pattern>/faces/listaKupionychBiletowKlient.xhtml</url-
pattern>
        <url-pattern>/faces/edytujMojeKonto.xhtml</url-pattern>
        <url-pattern>/faces/kupBilet.xhtml</url-pattern>
        <url-pattern>/faces/zwrocBilet.xhtml</url-pattern>
        <url-pattern>/faces/zmienMojeHaslo.xhtml</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <description/>
        <role-name>Klient</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>TicketShopJDBCRealm</realm-name>
    <form-login-config>
        <form-login-page>/faces/logowanie.xhtml</form-login-page>
        <form-error-page>/faces/error.xhtml</form-error-page>
    </form-login-config>
</login-config>

<security-role>
    <description/>
    <role-name>Administrator</role-name>
</security-role>
<security-role>
    <description/>
    <role-name>Pracownik</role-name>
</security-role>
<security-role>
    <description/>
    <role-name>Klient</role-name>
</security-role>

<error-page>
    <error-code>403</error-code>
    <location>/faces/error403.xhtml</location>
</error-page>
<error-page>
    <error-code>404</error-code>
    <location>/faces/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>
</web-app>

```

Listing 35: Fragment deskryptora wdrożenia aplikacji web.xml

Wzorce haseł w bazie są przechowywane jako skróty wyliczone algorytmem funkcji jednokierunkowej SHA256. Na listingu 36 została przedstawiona metoda setHasło ustawiająca hasło w bazie danych.

```

    public void setHasło(String hasło) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] encodedhash =
digest.digest(hasło.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.hasło = stringBuffer.toString();
        } catch (NoSuchAlgorithmException ex) {
            Logger.getLogger(Konto.class.getName()).log(Level.SEVERE, null,
ex);
        }
    }
}

```

Listing 36: Fragment kodu źródłowego metody setHasło klasy encyjnej Konto

3.3.5 Walidacja danych

Walidacja danych nie dopuszcza do wprowadzenia danych niespójnych, nie spełniających ustalonych formatów oraz reguł biznesowych. W aplikacji weryfikacja prowadzona jest w zakresie poprawności składniowej, semantycznej oraz biznesowej. Na listingu 37 został przedstawiony przykład kodu na walidację danych hasła podczas rejestracji konta użytkownika. Reguły walidacji wymagają aby hasło było złożone z nie mniej niż 8 znaków, w tym przynajmniej jedna wielka litera, cyfra oraz znak specjalny. Walidacja odbywa się na warstwie prezentacji, co pozwala na poprawne zapisanie danych w bazie bez dodatkowego obciążenia serwera aplikacji. W przypadku nie spełnienia warunków walidacji, aplikacja wyświetli użytkownikowi stosowny komunikat błędu zaprezentowany na rysunku 25 a dane nie zostaną zapisane w bazie.

```

#{msg['strona.rejestracja.konto.label.haslo']}*:
<h:inputSecret id="haslo" value="${rejestracjaKontaPageBean.kontoDTO.haslo}"
required="true" size="50" maxLength="64"
validatorMessage="#{msg['strona.rejestracja.konto.formularz.walidacja.haslo']}"
" requiredMessage="#{msg['strona.rejestracja.konto.formularz.haslo']}">
<f:validateRegex pattern="(?!.*\d)(?!.*[a-z])(?!.*[A-Z])(?!.*[!@#$%]).
{6,20})" />
</h:inputSecret>

```

Listing 37: Fragment kodu źródłowego strony JSF rejestracjaKonta.xhtml

Strona główna
Lista dostępnych koncertów
!BRAK UWIERZYTELNIENIA!
Utwórz konto
Zaloguj
Przypomnij hasło

- Niepoprawny format wprowadzonych danych: hasło powinno składać się z minimum 8 znaków w tym z przynajmniej jednej wielkiej i jednej małej litery (bez polskich znaków), przynajmniej jednej cyfry oraz znaku specjalnego spośród @\$%!

Rejestruj swoje konto

Login*:	klient10
Hasło*:	
Powtórz hasło*:	
Imię*:	Paulina
Nazwisko*:	Kubiak
E-mail*:	paulina@tss.com
Pytanie kontrolne*:	Imię matki?
Odpowiedz*:	Ewa

Rejestruj swoje konto
Anuluj

Rysunek 25: Przykład komunikatu błędu wynikającego z naruszenia reguł walidacji.

3.3.6 Obsługa błędów

Wyjątek zgłoszony w warstwie logiki biznesowej zostaje propagowany do warstwy prezentacji, dzięki deklaracji *throw*. Propagacja wyjątku do warstwy prezentacji umożliwia wyświetlenie stosownych komunikatów o zaistniałym błędzie w interfejsie użytkownika. Przykład obsługi błędów w warstwie prezentacji został pokazany w listingu 2.

Obsługa błędów w warstwie ziaren CDI sprowadza się do wyświetlania stosownych komunikatów informujących użytkownika o błędzie oraz odnotowania w dziennikach zdarzeń o wystąpieniu wyjątku. Przykład wyświetlanych komunikatów błędów napotkanych podczas próby zakupu biletu został zaprezentowany na rysunku 26.

Strona główna
klient
Klient
Wyloguj

- Do daty koncertu pozostało mniej niż 24h. Niestety nie można kupić już biletu pomimo, że w puli biletów pozostało:
- 1000

Wykonawca: Kult
Data [RRRR-MM-DD HH:mm]: Sun Nov 10 20:00:00 CET 2019
Miejsce:Dekompresja
Cena biletu [zł]:100
ilość biletów: 2

Kup bilet
Anuluj

Strona główna
klient
Klient
Wyloguj

- Wartość nie może być równa bądź mniejsza od zera. W puli dostępnych biletów pozostało:
- 22

Wykonawca: Bajm
Data [RRRR-MM-DD HH:mm]: Tue Dec 10 20:00:00 CET 2019
Miejsce:Wytwórnia
Cena biletu [zł]:50
ilość biletów: -3

Kup bilet
Anuluj

Strona główna
klient
Klient
Wyloguj

- Niestety podana ilość biletów jest już niedostępna. W puli dostępnych biletów pozostało:
- 10

Wykonawca: The Adicts
Data [RRRR-MM-DD HH:mm]: Tue Dec 24 20:00:00 CET 2019
Miejsce:Dekompresja
Cena biletu [zł]:10
ilość biletów: 11

Kup bilet
Anuluj

Rysunek 26: Przykład komunikatu błędu

3.4 Instrukcja wdrożenia

Do uruchomienia aplikacji niezbędne są:

- system operacyjny
- przeglądarka internetowa
- serwer Payara
- Java DB

Zakładając, że serwer Payara oraz Java DB są uruchomione należy wykonać poniższe instrukcje:

1. **Utworzenie bazy danych** zaleca się poprzez użycie programu narzędziowego `ij` dostarczonego wraz z Apache Derby. Na listingu 38 zaprezentowana została konfiguracja bazy danych.

Bazę danych o nazwie `jdbc:derby://localhost:1527/TicketSopSecond` należy utworzyć w katalogu z bazami danych zlokalizowanym w katalogu domowym używając polecenia `connect`, a następnie ustawić `create` na `true`, jak zaprezentowano na listingu 39, dla domyślnego dla Derby katalogu `.netbeans-derby`. Przykład tworzenia bazy danych został zaprezentowany w listingu 39.

```
<property name="URL"
value="jdbc:derby://localhost:1527/TicketSopSecond"/>
<property name="serverName" value="localhost"/>
<property name="PortNumber" value="1527"/>
<property name="DatabaseName" value="TicketSopSecond"/>
<property name="User" value="shop"/>
<property name="Password" value="shop"/>
</jdbc-connection-pool>
```

Listing 38: Fragment pliku konfiguracyjnego `glassfish-resource.xml`

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

Listing 39: Przykład tworzenia bazy danych JavaDB dla systemu operacyjnego Linux przy pomocy programu narzędziowego `ij`

2. Załadowanie pliku struktur baz danych oraz danych inicjujących

Listing 39 przedstawia wzór jak należy wgrać struktury baz danych z pliku *createDB.sql* poleceniem *run*. Następnie dane inicjujące z pliku *initDB.sql* również wgrać poleceniem *run*. Pliki znajdują się w katalogu *TicketShopSecond/src/main/resources/**.

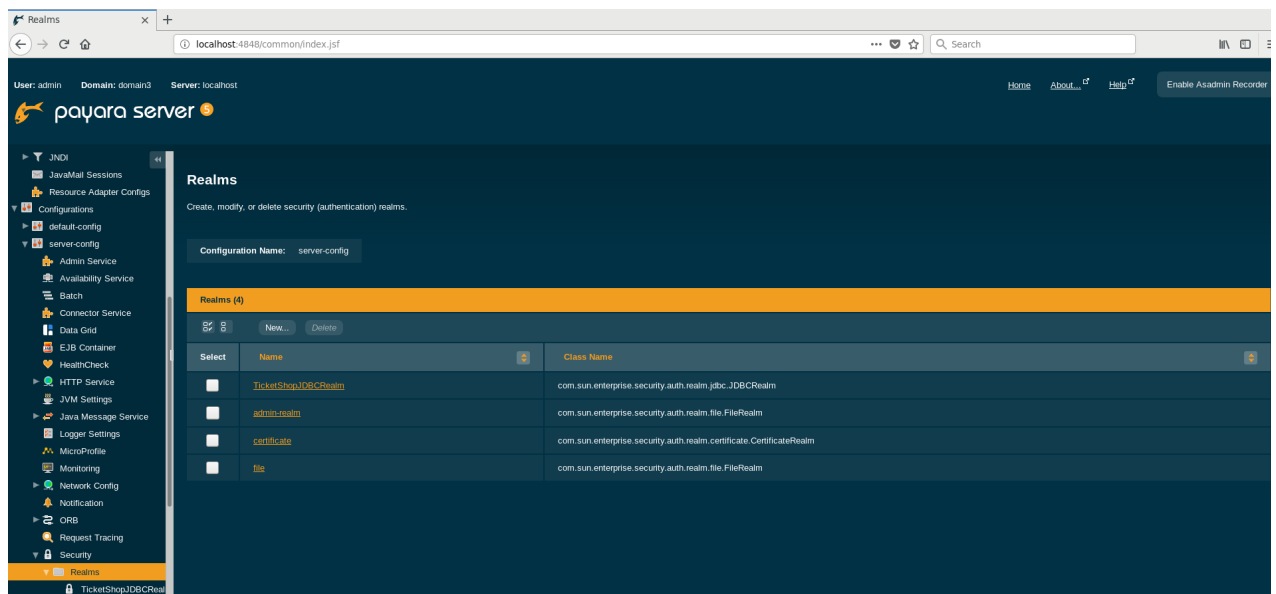
3. Nawiązanie połączenia z bazą danych

Po wykonaniu opisanych wcześniej czynności należy poleceniem *connect* połączyć się z utworzoną wcześniej bazą danych. W tym celu należy podać użytkownika oraz hasło, zgodnie z wartościami podanymi jako wartości *User* i *Password*, które zostały zaprezentowane w listingu 38.

4. Konfiguracja obszaru bezpieczeństwa na serwerze

Otworzyć przeglądarkę internetową i w pasku adres wpisać adres URL lokalnego serwera <http://localhost:4848/common/index.jsf>.

Z menu po lewej stronie należy wybrać: *Configurations* → *server-config* → *Security* → *Realms*. Następnie należy wybrać przycisk *New* znajdujący się po prawej stronie, który został zaprezentowany na rysunku 27.



Rysunek 27: Tworzenie Security Realm na serwerze aplikacyjnym Payara

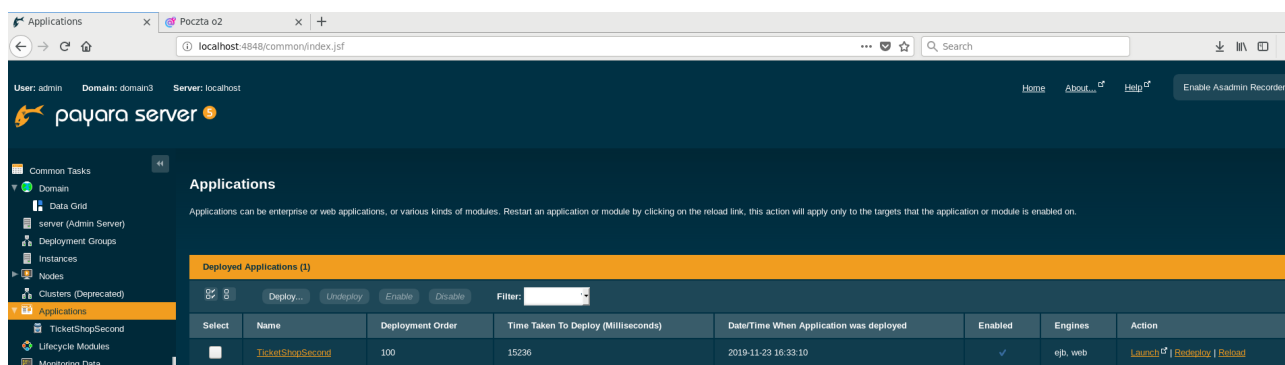
Konfigurację obszaru bezpieczeństwa przedstawia rysunek 28, na którym został zaprezentowany wzór wypełnionego poprawnie formularza. Po wpisaniu w odpowiednie miejsca danych należy zatwierdzić utworzenie *Security Realm* klikając na przycisk *OK*.

JAAS Context: *	jdbcRealm	Identifier for the login module to use for this realm
JNDI: *	jdbc/TicketShopDS	JNDI name of the JDBC resource used by this realm
User Table: *	AUTORYZACJA_VIEW	Name of the database table that contains the list of authorized users for this realm
User Name Column: *	LOGIN	Name of the column in the user table that contains the list of user names
Password Column: *	HASLO	Name of the column in the user table that contains the user passwords
Group Table: *	AUTORYZACJA_VIEW	Name of the database table that contains the list of groups for this realm
Group Table User Name Column:		Name of the column in the user group table that contains the list of groups for this realm
Group Name Column: *	KONTO	Name of the column in the group table that contains the list of group names
Assign Groups:		Comma-separated list of group names
Database User:		Specify the database user name in the realm instead of the JDBC connection pool
Database Password:		Specify the database password in the realm instead of the JDBC connection pool
Digest Algorithm:	SHA-256	Digest algorithm (default is SHA-256); note that the default was MD5 in GlassFish versions prior to 3.1
Encoding:		

Rysunek 28: Security Realm – konfiguracja na serwerze aplikacyjnym Payara

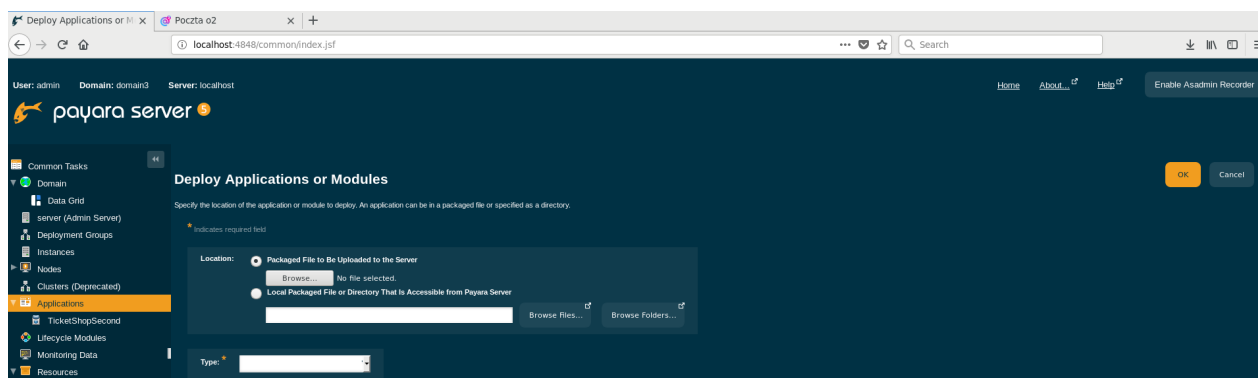
5. Wgranie aplikacji na serwer aplikacyjny i uruchomienie aplikacji

Z menu po lewej stronie należy wybrać przycisk *Application*, następnie wybrać przycisk *Deploy*, jak zaprezentowano na rysunku 29.



Rysunek 29: Wgranie aplikacji na serwer aplikacyjny Payara (krok pierwszy)

Poprzez użycie guzika *Browse* wybrać plik *TicketShopSecond-1.0.war*, który należy pobrać z załączonego do raportu nośnika. Ścieżka do pliku to *TicketShopSecond/target/TicketShopSecond-1.0.war*. Po wybraniu pliku należy wybrać przycisk *Otwórz* w oknie na dole strony oraz zatwierdzić wybór OK. Widok strony WWW serwera aplikacyjnego został zaprezentowany w drugim kroku na rysunku 30.



Rysunek 30: Wgranie aplikacji na serwer aplikacyjny Payara (krok drugi)

Aplikacja powinna być widoczna na serwerze. Aby ją uruchomić należy wybrać z kolumny *Action* przycisk *Launch*.

Jeśli wszystkie opisane w rozdziale punkty zostały wykonane poprawnie możliwe będzie zalogowanie się do aplikacji na jedno z predefiniowanych kont, znajdujących się w pliku z danymi inicjującymi. Loginy oraz hasła niezbędne do uwierzytelnienia zostały udostępnione w tabeli 2.

Tabela 2. Dane uwierzytelnienia do utworzonego systemu

Poziom dostępu	Login	Hasło
Administrator	admin	Qwertyuiop1!
Pracownik	pracownik	Qwertyuiop1!
Klient	klient	Qwertyuiop1!

3.5 Podsumowanie

Celem niniejszej pracy była implementacja biznesowego systemu służącego do sprzedaży biletów na koncerty. Zbudowany system informatyczny składa się z aplikacji internetowej oraz relacyjnej bazy danych. Aplikacja została wykonana przy użyciu technologii Java Enterprise Edition, osadzona w lokalnym serwerze aplikacyjnym Payara a relacyjna baza danych została zlokalizowana w lokalnym systemie zarządzania bazami danych JavaDB.

Architektura systemu została wykonana w modelu trójwarstwowym. Zostały od siebie odseparowane warstwy widoku, logiki biznesowej oraz składowania danych. Logika biznesowa została zaimplementowana w stanowych i bezstanowych komponentach EJB. Wzorzec DTO separuje od siebie warstwę prezentacji od warstwy składowania danych. Dzięki wykorzystaniu obiektów DTO do transportu danych pomiędzy warstwą widoku a warstwą logiki biznesowej, czyli odpowiednio pomiędzy ziarnami CDI a komponentami EJB, mamy większą kontrolę nad tym co jest udostępniane użytkownikowi. Odczyt danych z bazy jest realizowany z wykorzystaniem encji JPA. Również utrwalenie danych biznesowych w relacyjnej bazie danych zostało osiągnięte stosując standard mapowania obiektowo-relacyjnego w postaci encji JPA. W warstwie prezentacji zostały wykorzystane ziarna CDI oraz technologię JSF. Zastosowanie takiej architektury okazało się dobrym wyborem ze względu na dużą elastyczność w implementacji.

Zbudowany system jest wielodostępny i zgodnie z przyjętymi założeniami zostały wyodrębnione cztery poziomy dostępu dla kont użytkowników systemu. Są to:

- Gość, osoba odwiedzająca stronę w celu zapoznania się z ofertą sklepu;
- Administrator, osoba zajmująca się obsługą kont użytkowników;
- Pracownik, osoba zatrudniona przez sklep w celu obsługi koncertów oraz klientów;
- Klient, kontrahent dokonujący zakupu;

Dostęp do poszczególnych funkcjonalności oraz elementów widoku staje się możliwy poprzez uwierzytelnienie na wcześniej założonym koncie. Autoryzacji użytkownika dokonuje Administrator przypisując do zarejestrowanego utworzonego konta poziom dostępu. Wyjątkiem jest poziom Gość, który to jest użytkownikiem niepotwierdzonym i nie wymaga uwierzytelnienia. Po zalogowaniu do systemu użytkownik posiada odrębny widok oraz ograniczony dostęp do funkcjonalności systemu zdefiniowany dla ról.

Dla wszystkich elementów widoku został zapewniony ujednolicony wygląd graficznego interfejsu użytkownika w oparciu o Faceletes Template dostarczane przez JSF. Dostęp do aplikacji jest możliwy poprzez interfejs użytkownika, który wymaga wykorzystania współczesnej przeglądarki internetowej. Użytkownik może korzystać z interfejsu użytkownika stworzonego systemu w polskiej oraz angielskojęzycznej wersji językowej według ustawionych w przeglądarce internetowej preferencji. W warstwie prezentacji został

zaimplementowany mechanizm walidacji danych, dzięki któremu użytkownik dostaje czytelny komunikat o ograniczeniach oraz wymaganiach w wypełnianym formularzu.

Aplikacja została zaimplementowana tak aby móc obsłużyć wielu użytkowników w tym samym czasie. W związku z tym została zapewniona ochrona spójności danych, dzięki przetwarzaniu transakcyjnemu (ang. *On-line Transactional Proccesing*) oraz zastosowaniu mechanizmu blokad optymistycznych. Transakcje w zbudowanym systemie informatycznym spełniają warunki, określane w literaturze akronimem ACID (ang. *Atomicity, Consistency, Isolation, Durability*). Metody biznesowe wykorzystują mechanizm zarządzania granicami transakcji przez komponenty EJB, w strategii CMT (ang. *Container Menagment Transactions*).

Został wykorzystany mechanizm bezpieczeństwa, który kontroluje dostęp użytkowników do metod biznesowych oraz danych w związku z przyznanym dla ich konta poziomem dostępu poprzez wykorzystanie adnotacji `@RolesAllowed`.

Każde zgłoszone przez system zdarzenie związane z wykonywaniem akcji przez użytkowników systemu jest zapisywane w dziennikach zdarzeń. W dziennikach zdarzeń również zapisywane są błędy oraz granice transakcji aplikacyjnych. Wykorzystano w tym celu implementację mechanizmu obiektu przechwytyjącego, czyli inceptora, który jest powiązany z metodami biznesowymi komponentów EJB. Rejestrowanie granic transakcji jest zapewnione poprzez zaimplementowany interfejs `SessionSynchronization`. Dodatkowo system przechowuje w bazie danych informacje o dacie, godzinie oraz użytkowniku, który dokonał ostatniej modyfikacji rekordu.

W aplikacji został wykorzystany system obsługujący błędy aplikacyjne. W przypadku naruszenia reguł biznesowych, zostaje zgłoszony wyjątek, który następnie jest propagowany do warstwy prezentacji, oraz wyświetlany użytkownikowi w postaci komunikatu.

Reasumując, postawiony cel pracy został osiągnięty. Zbudowany system informatyczny spełnia wszystkie przyjęte założenia funkcjonalne oraz нефункционалне. W przyszłości programista może rozwijać aplikację o dodatkowe funkcjonalności oraz wymagania stawiane przez klienta. Modułami o jakie mogłabym rozszerzyć powstałą aplikację są płatność elektroniczna za zakupiony bilet oraz powiadomienia wiadomością e-mail o dokonanych zakupie.

4 Źródła

Bibliografia

1. Anghel Leonard: JavaServer Faces 2.2. Mistrzowskie programowanie, Wydanie , , 2016
2. Christian Bauer, Gavin King, Gary Gregory: Java Persistence. Programowanie aplikacji bazodanowych w Hibernate. Wydanie II, Wydanie , Helion, 2016
3. Murat Yener, Alex Theedom: Java EE. Zaawansowane wzorce projektowe, Wydanie , , 2015
4. Krzysztof Rychlicki-Kicior: Java EE 6. Programowanie aplikacji WWW. Wydanie IIJava EE 6. Programowanie aplika, Wydanie II, , 2015
5. : URL: THE WORLD'S LARGEST WEB DEVELOPER SITE,
https://www.w3schools.com/bootstrap/bootstrap_get_started.asp (Dostęp: sierpień 2019),

5 Spis załączników

Wykaz rysunków

Rysunek 1: Przypadki użycia dla poziomu dostępu Gość.....	4
Rysunek 2: Przypadki użycia dla poziomu dostępu Klient.....	5
Rysunek 3: Przypadki użycia dla poziomu dostępu Pracownik.....	5
Rysunek 4: Przypadki użycia dla poziomu dostępu Administrator.....	6
Rysunek 5: Formularz rejestracji nowego konta.....	7
Rysunek 6: Lista kont z przypisanym poziomem dostępu, funkcjonalność aktywuj oraz dezaktywuj wybrane konto.....	8
Rysunek 7: Zmiana swojego hasła przy pomocy pytania kontrolnego.....	8
Rysunek 8: Formularz logowanie do systemu.....	9
Rysunek 9: Wylogowanie z systemu.....	9
Rysunek 10: Formularz edycji danych swojego konta (na przykładzie konta Administrator).....	9
Rysunek 11: Formularz zmiany hasła dla zalogowanego użytkownika.....	10
Rysunek 12: Formularz zmiany hasła przez administratora.....	10
Rysunek 13: Usuwanie konta.....	11
Rysunek 14: Ustawienie poziomu dostępu.....	11
Rysunek 15: Lista wszystkich kupionych biletów (widok dla pracownika).....	12
Rysunek 16: Lista kupionych biletów (widok dla klienta).....	12
Rysunek 17: Formularz zakupu biletu.....	13
Rysunek 18: Formularz rejestracji koncertu.....	13
Rysunek 19: Formularz edycji koncertu.....	13
Rysunek 20: Lista dostępnych koncertów (widok dla klienta).....	14
Rysunek 21: Usuń koncert.....	14
Rysunek 22: Diagram komponentów – utwórz konto.....	15
Rysunek 23: Model bazy danych.....	27
Rysunek 24: Diagram klas encyjných.....	29
Rysunek 25: Przykład komunikatu błędu wynikającego z naruszenia reguł walidacji.....	41
Rysunek 26: Przykład komunikatu błędu.....	41
Rysunek 27: Tworzenie Security Realm na serwerze aplikacyjnym Payara.....	43
Rysunek 28: Security Realm – konfiguracja na serwerze aplikacyjnym Payara.....	44
Rysunek 29: Wgranie aplikacji na serwer aplikacyjny Payara (krok pierwszy).....	44
Rysunek 30: Wgranie aplikacji na serwer aplikacyjny Payara (krok drugi).....	45

Wykaz listingów

Listing 1: Fragment kodu źródłowego strony JSF rejestracjaKonta.xhtml.....	16
Listing 2: Ziarno CDI RejestracjaKontaPageBean.....	17
Listing 3: Ziarno CDI RejestracjaKontaControllerBean.....	17
Listing 4: Fragment kodu źródłowego klasy KontoEndpoint.....	18
Listing 5: Metoda create klasy AbstractFacade.....	18
Listing 6: Fragment kodu źródłowego strony JSF listaRejestracjiKonta.xhtml.....	19
Listing 7: Fragment kodu klasy ListaKontPageBean.....	20
Listing 8: Fragment kodu klasy KontoEndpoint.....	20
Listing 9: Fragment kodu klasy KontoFacade.....	21
Listing 10: Kwerenda wyszukująca nowe konta z klasy encyjných Konto.....	21

Listing 11: Fragment kodu strony JSF edytujKonto.xhtml.....	21
Listing 12: Fragment kodu klasy EdycjaKontaPageBean.....	22
Listing 13: Fragment kodu klasy RejestracjaKontaControllerBean.....	23
Listing 14: Fragment kodu klasy KontoEndpoint.....	23
Listing 15: Fragment kodu klasy KontoFacade.....	23
Listing 16: Fragment kodu klasy usunKonto.xhtml.....	24
Listing 17: Fragment kodu klasy UsunKontoPageBean.....	24
Listing 18: Fragment kodu klasy RejestracjaKontaControllerBean.....	25
Listing 19: Fragment kodu klasy KontoEndpoint.....	25
Listing 20: Fragment kodu klasy KontoFacade.....	26
Listing 21: Kwerenda wyszukiwająca konta po loginie z klasy encyjnej Konto.....	26
Listing 22: Zawartość pliku konfiguracyjnego persistence.xml.....	28
Listing 23: Fragment kodu klasy bazowej AbstractEntity.....	29
Listing 24: Fragment kodu klasy encyjnej Konto.....	30
Listing 25: Implementacja klasy bazowej AbstractEndpoint.....	31
Listing 26: Implementacja bazowej klasy AbstractFacade.....	32
Listing 27: Kod klasy LoggingInterceptor.....	32
Listing 28: Fragment deskryptora wdrożenia glassfish-resources.xml.....	33
Listing 29: Plik konfiguracyjny glassfish-web.xml.....	34
Listing 30: Fragment kodu źródłowego klasy bazowej AppBaseException.....	35
Listing 31: Fragment kodu źródłowego klasy błędów aplikacyjnych BiletException.....	36
Listing 32: Fragment kodu klasy BiletDTO.....	37
Listing 33: Konfiguracja w deskrytorze wdrożenia web.xml.....	37
Listing 34: Fragment kodu strony JSF logowanie.xhtml.....	38
Listing 35: Fragment deskryptora wdrożenia aplikacji web.xml.....	40
Listing 36: Fragment kodu źródłowego metody setHaslo klasy encyjnej Konto.....	40
Listing 37: Fragment kodu źródłowego strony JSF rejestracjaKonta.xhtml.....	40
Listing 38: Fragment pliku konfiguracyjnego glassfish-resource.xml.....	42
Listing 39: Przykład tworzenia bazy danych JavaDB dla systemu operacyjnego Linux przy pomocy programu narzędziowego ij.....	42