Bazy danych	Dokumentacja projektu
Autor	Michał Dyjak, 125114
Kierunek, rok	Informatyka, II rok, st. stacjonarne (3,5-l)
Temat projektu	*Aplikacja do zarządzania zasobami klientów banku.*

Wstęp

Opis aplikacji

Gigabank Administrator Panel to zaawansowana aplikacja desktopowa zaprojektowana specjalnie dla administratorów banku. Jej głównym celem jest ułatwienie zarządzania wszystkimi aspektami kont klientów poprzez zapewnienie pełnej funkcjonalności CRUD (Create, Read, Update, Delete). Aplikacja umożliwia administratorom wykonywanie wszelkich operacji związanych z zarządzaniem tabelami użytkowników, kont bankowych, transakcji, depozytów, pożyczek oraz walut, co przyczynia się do efektywnego i bezpiecznego zarządzania zasobami finansowymi banku.

Przeznaczenie

Aplikacja została stworzona z myślą o pracownikach banku pełniących funkcje administracyjne. Dzięki niej mogą oni w sposób zorganizowany i bezpieczny przeprowadzać operacje na kontach klientów oraz zarządzać innymi istotnymi elementami bankowego systemu informacyjnego. Gigabank Administrator Panel jest narzędziem, które pozwala na:

1. **Zarządzanie użytkownikami (users):**
    - Administratorzy mogą tworzyć nowe konta użytkowników, przeglądać szczegóły istniejących kont, aktualizować informacje oraz usuwać konta użytkowników, zapewniając odpowiednie uprawnienia i dostępy.
2. **Zarządzanie kontami bankowymi (accounts):**
    - Aplikacja umożliwia tworzenie, odczyt, aktualizację i usuwanie kont bankowych. Administratorzy mają dostęp do szczegółowych informacji o

kontach, takich jak login, uprawnienia, saldo, historia transakcji, liczba depozytów, liczba pożyczek oraz data utworzenia konta.

3. **Zarządzanie transakcjami (transactions):**
    - Administratorzy mogą przeglądać historię transakcji związanych z kontami bankowymi, a także dodawać, modyfikować i usuwać transakcje. Umożliwia to dokładne śledzenie przepływów finansowych i zapewnienie zgodności z wewnętrznymi politykami banku oraz regulacjami prawnymi.
4. **Zarządzanie depozytami (deposits):**
    - Aplikacja umożliwia zarządzanie depozytami klientów, w tym tworzenie nowych depozytów, przeglądanie szczegółów istniejących depozytów, aktualizację warunków depozytów oraz ich usuwanie w razie potrzeby.
5. **Zarządzanie pożyczkami (loans):**
    - Administratorzy mogą tworzyć nowe pożyczki, przeglądać szczegóły istniejących pożyczek, aktualizować warunki oraz zarządzać spłatami pożyczek. Aplikacja wspiera pełny cykl życia pożyczki, od jej utworzenia po finalizację.
6. **Zarządzanie walutami (currencies):**
    - Aplikacja pozwala na zarządzanie walutami, które mogą być używane w banku. Administratorzy mogą dodawać nowe waluty, aktualizować kursy wymiany, usuwać nieaktywne waluty oraz przeglądać listę wszystkich dostępnych walut. Zarządzanie walutami jest kluczowe dla obsługi kont wielowalutowych oraz transakcji międzynarodowych.

Technologie użyte w aplikacji

Gigabank Administrator Panel został stworzony przy użyciu najnowocześniejszych technologii, które zapewniają wysoką wydajność, bezpieczeństwo i łatwość w zarządzaniu bankowymi zasobami. Poniżej znajduje się szczegółowy opis kluczowych technologii użytych w tej aplikacji:

JavaFX

JavaFX to nowoczesna platforma służąca do tworzenia aplikacji graficznych (GUI) w języku Java. Gigabank Administrator Panel wykorzystuje JavaFX do budowy interfejsu użytkownika, który jest intuicyjny, responsywny i estetyczny. JavaFX oferuje szereg komponentów, takich jak przyciski, pola tekstowe, tabele i panele, które umożliwiają tworzenie bogatych interfejsów użytkownika. Dzięki temu, użytkownicy mogą w łatwy sposób nawigować po aplikacji i wykonywać potrzebne operacje.

JDBC (Java Database Connectivity)

Java Database Connectivity (JDBC) to API w języku Java, które umożliwia łączenie się z bazami danych, wykonywanie zapytań i zarządzanie danymi. Gigabank Administrator Panel wykorzystuje JDBC do komunikacji z bazą danych Oracle. JDBC zapewnia bezpieczne i efektywne połączenie, umożliwiając administratorom wykonywanie operacji CRUD na różnych tabelach, takich jak użytkownicy, konta, transakcje, depozyty, pożyczki oraz waluty. Dzięki JDBC, aplikacja może w łatwy sposób wykonywać złożone operacje bazodanowe i zarządzać danymi w czasie rzeczywistym.

Oracle Database 23c

Oracle Database 23c to nowoczesny system zarządzania bazą danych, który oferuje zaawansowane funkcje przechowywania, zarządzania i analizy danych. Gigabank Administrator Panel korzysta z Oracle Database 23c jako lokalnego serwera bazodanowego. Oracle Database zapewnia wysoką wydajność, skalowalność i bezpieczeństwo danych, co jest kluczowe dla aplikacji bankowych. Dzięki temu, aplikacja może przechowywać i zarządzać dużymi ilościami danych klientów, transakcji, depozytów i innych istotnych informacji w sposób niezawodny i bezpieczny.

PL/SQL (Procedural Language/Structured Query Language)

PL/SQL to rozszerzenie języka SQL opracowane przez firmę Oracle, które umożliwia tworzenie procedur, funkcji i bloków kodu do wykonywania złożonych operacji na bazie danych. Gigabank Administrator Panel wykorzystuje PL/SQL do realizacji wszystkich operacji na bazie danych. PL/SQL pozwala na tworzenie złożonych zapytań i operacji, które są wykonywane bezpośrednio na serwerze bazodanowym, co zwiększa wydajność i efektywność. Dzięki PL/SQL, aplikacja może wykonywać operacje takie jak tworzenie, odczyt, aktualizacja i usuwanie danych w sposób szybki i bezpieczny.

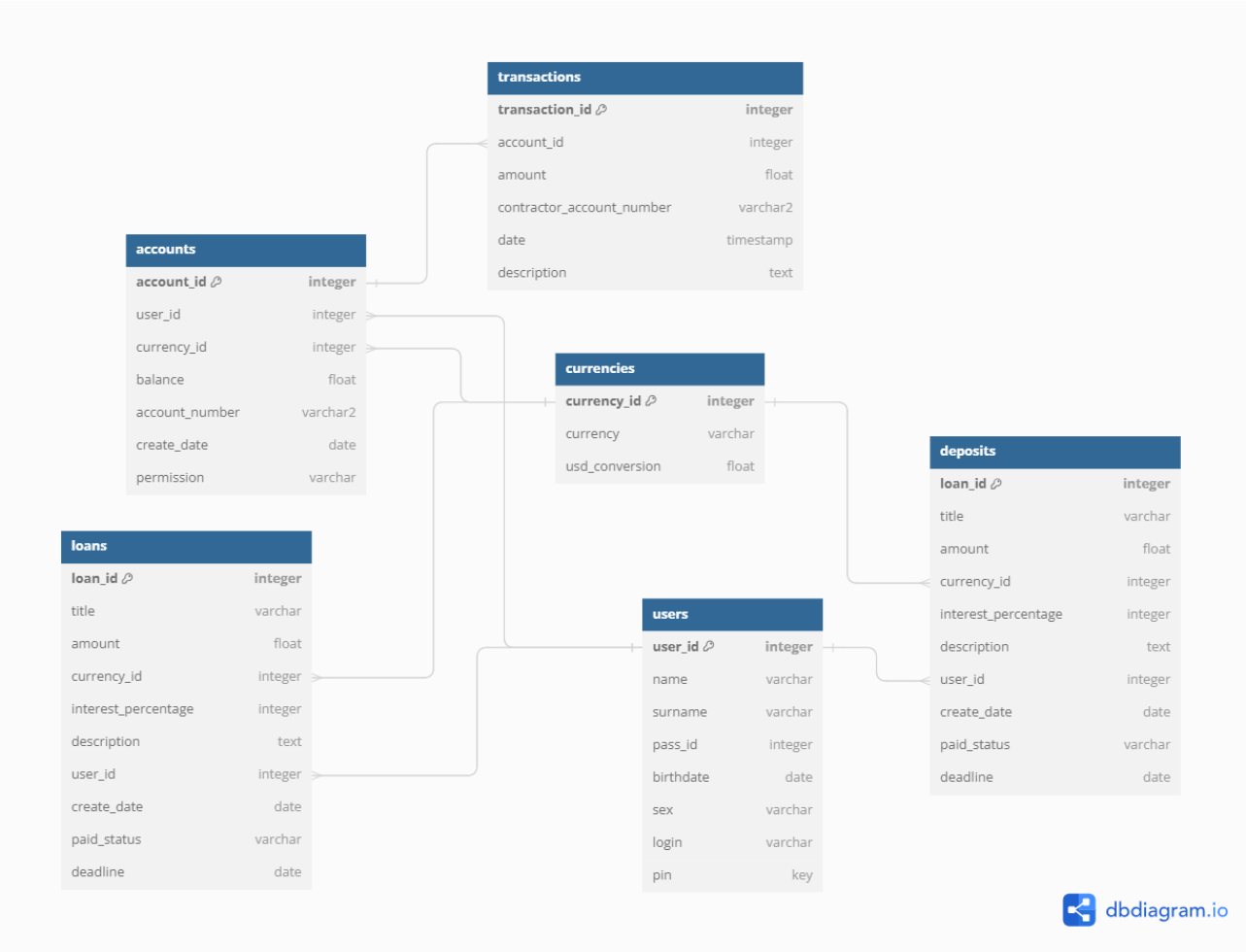
Inne technologie

7. **Java 17**: Gigabank Administrator Panel jest zbudowany przy użyciu Java 17, która jest najnowszą wersją języka Java. Java 17 oferuje liczne ulepszenia i nowe funkcje, które przyczyniają się do zwiększenia wydajności i bezpieczeństwa aplikacji.
8. **FXML**: Do definiowania interfejsu użytkownika użyto FXML, który jest formatem XML specyficznym dla JavaFX. FXML umożliwia oddzielenie logiki aplikacji od jej wyglądu, co przyczynia się do łatwiejszej konserwacji i rozwoju aplikacji.
9. **CSS**: Stylizacja interfejsu użytkownika jest zarządzana za pomocą CSS (Cascading Style Sheets). CSS pozwala na łatwe i elastyczne dostosowywanie

wyglądu aplikacji, co umożliwia tworzenie estetycznych i spójnych interfejsów użytkownika.

Opis bazy danych i procedur

Diagram ERD



Tabele

10. **USERS**:
    - **USER_ID**: Unikalny identyfikator użytkownika (klucz główny).
    - **NAME**: Imię użytkownika.
    - **SURNAME**: Nazwisko użytkownika.
    - **BIRTHDATE**: Data urodzenia użytkownika.
    - **SEX**: Płeć użytkownika.

- **LOGIN**: Login użytkownika.
- **PIN**: PIN użytkownika.

11. **TRANSACTIONS**:
    - **TRANSACTION_ID**: Unikalny identyfikator transakcji (klucz główny).
    - **ACCOUNT_ID**: Identyfikator konta (klucz obcy do tabeli ACCOUNTS).
    - **AMOUNT**: Kwota transakcji.
    - **CONTRACTOR_ACCOUNT_NUMBER**: Numer konta kontrahenta.
    - **EXECUTE_DATE**: Data wykonania transakcji.
    - **DESCRIPTION**: Opis transakcji.
12. **LOANS**:
    - **LOAN_ID**: Unikalny identyfikator pożyczki (klucz główny).
    - **TITLE**: Tytuł pożyczki.
    - **AMOUNT**: Kwota pożyczki.
    - **CURRENCY_ID**: Identyfikator waluty (klucz obcy do tabeli CURRENCIES).
    - **INTEREST_PERCENTAGE**: Procent odsetek.
    - **DESCRIPTION**: Opis pożyczki.
    - **USER_ID**: Identyfikator użytkownika (klucz obcy do tabeli USERS).
    - **CREATE_DATE**: Data utworzenia pożyczki.
    - **DEADLINE**: Termin spłaty pożyczki.
    - **STATUS**: Status pożyczki.
13. **DEPOSITS**:
    - **DEPOSIT_ID**: Unikalny identyfikator depozytu (klucz główny).
    - **TITLE**: Tytuł depozytu.
    - **AMOUNT**: Kwota depozytu.
    - **CURRENCY_ID**: Identyfikator waluty (klucz obcy do tabeli CURRENCIES).
    - **INTEREST_PERCENTAGE**: Procent odsetek.
    - **DESCRIPTION**: Opis depozytu.
    - **USER_ID**: Identyfikator użytkownika (klucz obcy do tabeli USERS).
    - **CREATE_DATE**: Data utworzenia depozytu.
    - **DEADLINE**: Termin zakończenia depozytu.
    - **STATUS**: Status depozytu.
14. **CURRENCIES**:
    - **CURRENCY_ID**: Unikalny identyfikator waluty (klucz główny).
    - **CURRENCY**: Nazwa waluty.
    - **USD_CONVERSION**: Kurs przeliczeniowy do USD.
15. **ACCOUNTS**:
    - **ACCOUNT_ID**: Unikalny identyfikator konta (klucz główny).
    - **USER_ID**: Identyfikator użytkownika (klucz obcy do tabeli USERS).
    - **CURRENCY_ID**: Identyfikator waluty (klucz obcy do tabeli CURRENCIES).

- **BALANCE**: Saldo konta.
- **ACCOUNT_NUMBER**: Numer konta.
- **CREATE_DATE**: Data utworzenia konta.
- **PERMISSION**: Uprawnienia do konta.

Sekwencje

- **ACCOUNTS_SEQ**: Sekwencja do generowania unikalnych identyfikatorów kont.
- **CURRENCY_ID_SEQ**: Sekwencja do generowania unikalnych identyfikatorów walut.
- **DEPOSIT_ID_SEQ**: Sekwencja do generowania unikalnych identyfikatorów depozytów.
- **LOANS_SEQ**: Sekwencja do generowania unikalnych identyfikatorów pożyczek.
- **TRANSACTIONS_SEQ**: Sekwencja do generowania unikalnych identyfikatorów transakcji.
- **USERS_SEQ**: Sekwencja do generowania unikalnych identyfikatorów użytkowników.

Triggery

16. **CURRENCIES_TRIGGER**:
    - Wyzwalacz przed wstawieniem nowej waluty, ustawiający `CURRENCY_ID` na następny numer sekwencji `currency_id_seq`.
17. **DEPOSIT_ID_TRIGGER**:
    - Wyzwalacz przed wstawieniem nowego depozytu, ustawiający `DEPOSIT_ID` na następny numer sekwencji `deposit_id_seq`.
18. **LOANS_TRIGGER**:
    - Wyzwalacz przed wstawieniem nowej pożyczki, ustawiający `LOAN_ID` na następny numer sekwencji `loans_seq`.
19. **TRG_TRANSACTIONS_BEFORE_INSERT**:
    - Wyzwalacz przed wstawieniem nowej transakcji, ustawiający `TRANSACTION_ID` na następny numer sekwencji `transactions_seq`.
20. **TRG_USERS_ID**:
    - Wyzwalacz przed wstawieniem nowego użytkownika, ustawiający `USER_ID` na następny numer sekwencji `users_seq`.

Typy

- **balance_currency**: Typ obiektowy przechowujący saldo (`balance` jako `FLOAT`) i walutę (`currency` jako VARCHAR2).

Procedury

21. **add_account**:
    - Dodaje nowe konto. Przyjmuje parametry: `p_user_id`, `p_currency_id`, `p_balance`, `p_account_number`, `p_create_date`, `p_permission`.
    - Sprawdza, czy saldo jest nieujemne.
22. **add_currency**:
    - Dodaje nową walutę. Przyjmuje parametry: `argument1` (nazwa waluty) i `argument2` (kurs do USD).
    - Sprawdza, czy kurs jest nieujemny.
23. **add_deposit**:
    - Dodaje nowy depozyt. Przyjmuje parametry: `p_title`, `p_amount`, `p_currency_id`, `p_interest_percentage`, `p_description`, `p_user_id`, `p_create_date`, `p_deadline`, `p_status`.
    - Sprawdza, czy kwota depozytu jest nieujemna.
24. **add_loan**:
    - Dodaje nową pożyczkę. Przyjmuje parametry: `p_title`, `p_amount`, `p_currency_id`, `p_interest_percentage`, `p_description`, `p_user_id`, `p_create_date`, `p_deadline`, `p_status`.
    - Sprawdza, czy kwota pożyczki jest nieujemna.
25. **add_transaction**:
    - Dodaje nową transakcję. Przyjmuje parametry: `p_account_id`, `p_amount`, `p_contractor_account_num`, `p_execute_date`, `p_description`.
26. **add_user**:
    - Dodaje nowego użytkownika. Przyjmuje parametry: `p_name`, `p_surname`, `p_birthdate`, `p_sex`, `p_login`, `p_pin`.
27. **delete_account**:
    - Usuwa konto na podstawie `p_account_id`.
    - Sprawdza, czy istnieją powiązane rekordy, które mogą uniemożliwić usunięcie.
28. **delete_currency**:
    - Usuwa walutę na podstawie `currency_id_in`.
29. **delete_deposit**:
    - Usuwa depozyt na podstawie `p_deposit_id`.
30. **delete_loan**:
    - Usuwa pożyczkę na podstawie `p_loan_id`.
31. **delete_transaction**:
    - Usuwa transakcję na podstawie `p_transaction_id`.
32. **delete_user**:

- Usuwa użytkownika na podstawie `p_user_id`.
- Sprawdza, czy istnieją powiązane rekordy, które mogą uniemożliwić usunięcie.

33. **REFRESHALLTABLES**:
    - Usuwa wszystkie rekordy z tabel, wyłącza sprawdzanie kluczy obcych, aktualizuje tabele i ponownie włącza sprawdzanie kluczy obcych.
34. **search_account_info**:
    - Szuka informacji o koncie na podstawie `p_account_id`.
    - Tworzy widok `account_info_view` zawierający informacje o właścicielu, uprawnieniach, numerze konta, saldzie, liczbie depozytów i pożyczek oraz dacie utworzenia konta.
35. **search_currencies**:
    - Szuka walut na podstawie `p_argument`.
    - Tworzy widok `temp_currencies` zawierający pasujące waluty.
36. **search_users**:
    - Szuka użytkowników na podstawie `argument1` i `argument2`.
    - Tworzy widok `temp_users` zawierający pasujących użytkowników.
37. **update_account**:
    - Aktualizuje konto na podstawie `p_account_id`. Przyjmuje parametry: `p_user_id`, `p_currency_id`, `p_balance`, `p_account_number`, `p_create_date`, `p_permission`.
    - Sprawdza, czy saldo jest nieujemne.
38. **update_currency**:
    - Aktualizuje walutę na podstawie `p_currency_id`. Przyjmuje parametry: `p_currency` i `p_usd_conversion`.
39. **update_deposit**:
    - Aktualizuje depozyt na podstawie `p_deposit_id`. Przyjmuje parametry: `p_title`, `p_amount`, `p_currency_id`, `p_interest_percentage`, `p_description`, `p_user_id`, `p_create_date`, `p_deadline`, `p_status`.
    - Sprawdza, czy kwota depozytu jest nieujemna.
40. **update_loan**:
    - Aktualizuje pożyczkę na podstawie `p_loan_id`. Przyjmuje parametry: `p_title`, `p_amount`, `p_currency_id`, `p_interest_percentage`, `p_description`, `p_user_id`, `p_create_date`, `p_deadline`, `p_status`.
    - Sprawdza, czy kwota pożyczki jest nieujemna.
41. **update_transaction**:
    - Aktualizuje transakcję na podstawie `p_transaction_id`. Przyjmuje parametry: `p_account_id`, `p_amount`, `p_contractor_account_num`, `p_execute_date`, `p_description`.
42. **update_user**:

- Aktualizuje użytkownika na podstawie `p_user_id`. Przyjmuje parametry: `p_name`, `p_surname`, `p_birthdate`, `p_sex`, `p_login`, `p_pin`.

Uwagi dotyczące wdrożenia

- Sekwencje są używane do automatycznego generowania unikalnych identyfikatorów dla kluczowych pól.
- Triggery zapewniają integralność danych, automatycznie ustawiając wartości identyfikatorów.
- Procedury umożliwiają zarządzanie bazą danych poprzez operacje CRUD (Create, Read, Update, Delete).
- Typ obiektowy `balance_currency` pozwala na bardziej złożone przechowywanie danych dotyczących salda i waluty.

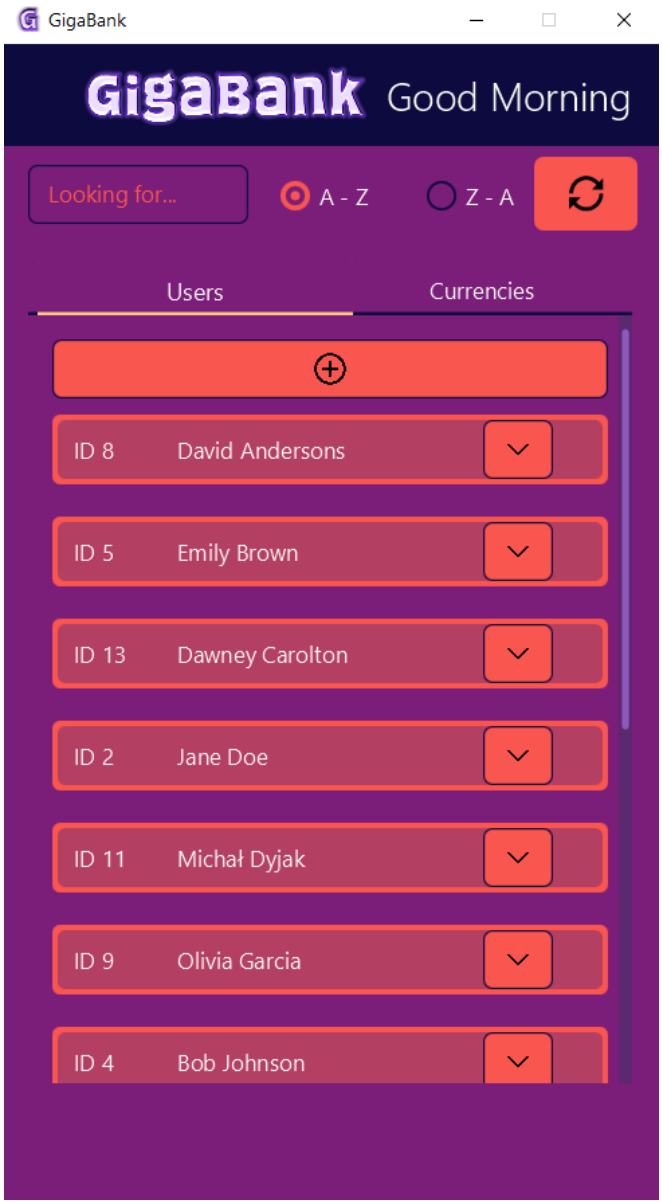
Dalsze kroki

- Wdrożenie skryptów SQL na serwerze Oracle.
- Testowanie funkcjonalności bazodanowej, w tym wywoływania procedur i triggerów.
- Optymalizacja oraz monitorowanie wydajności i integralności danych.

GUI

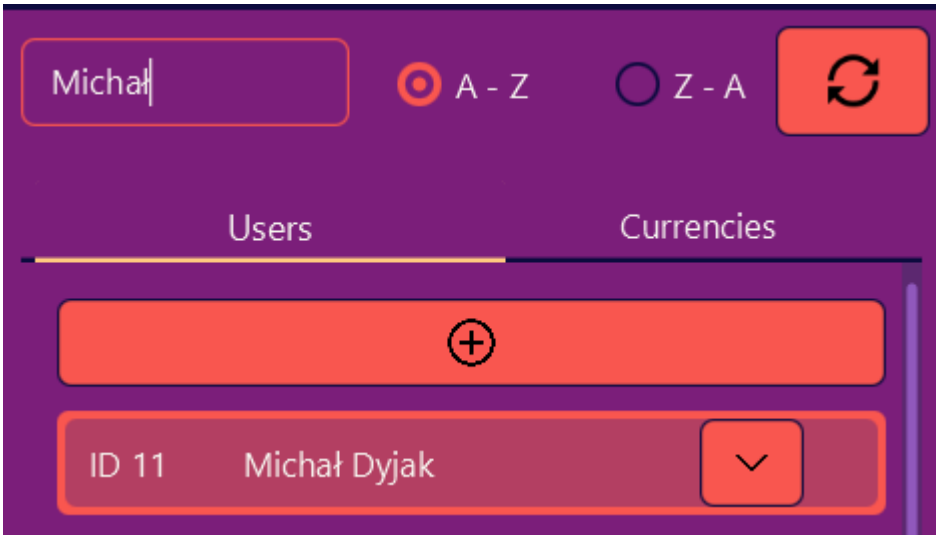
Aplikacja zarządzania bankowością została zaprojektowana z myślą o zapewnieniu użytkownikom wygodnego i intuicyjnego interfejsu do zarządzania danymi finansowymi. Głównym celem projektu jest umożliwienie łatwego dostępu do informacji o użytkownikach, ich kontach, transakcjach, lokatach oraz pożyczkach, a także zarządzanie walutami w systemie. Aplikacja oferuje pełen zakres operacji CRUD (tworzenie, odczyt, aktualizacja, usuwanie) z odpowiednimi formularzami i walidacją danych, aby zapewnić poprawność i integralność przechowywanych informacji.

Interfejs użytkownika został zaprojektowany z dbałością o estetykę i użyteczność, zapewniając przejrzysty układ elementów i przyjazną kolorystykę. Dzięki temu użytkownicy mogą łatwo poruszać się po aplikacji, szybko odnajdywać potrzebne informacje oraz efektywnie zarządzać danymi bankowymi.



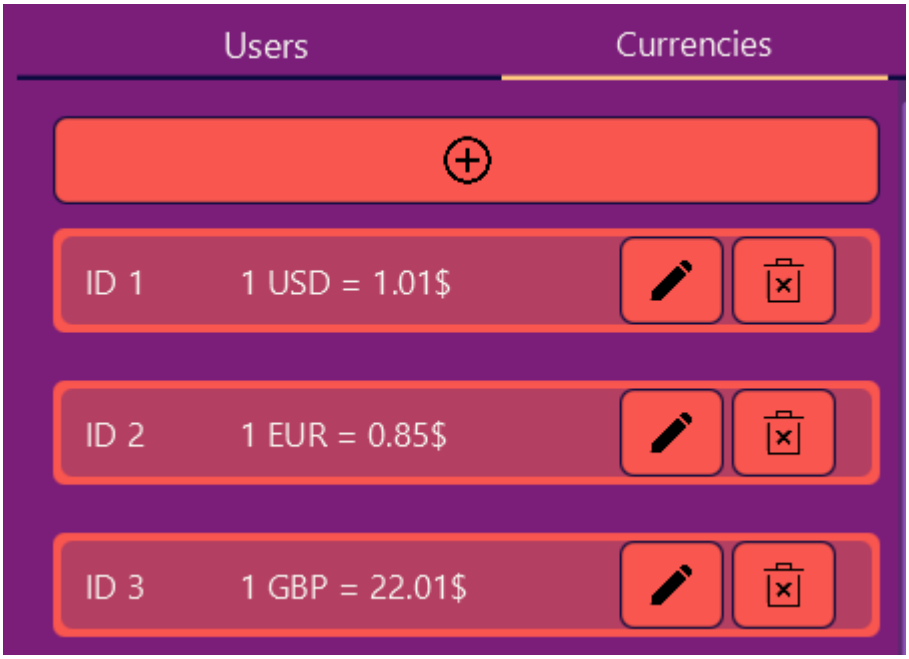
1. Panel Kontrolek

- **Pasek wyszukiwania po nazwie**: Umożliwia wyszukiwanie użytkowników według nazwiska.
- **Opcje sortowania po nazwie**: Umożliwia sortowanie użytkowników alfabetycznie.
- **Przycisk odświeżający**: Odświeża wyniki, pobierając najnowsze dane z bazy danych.



2. Panel z Walutami

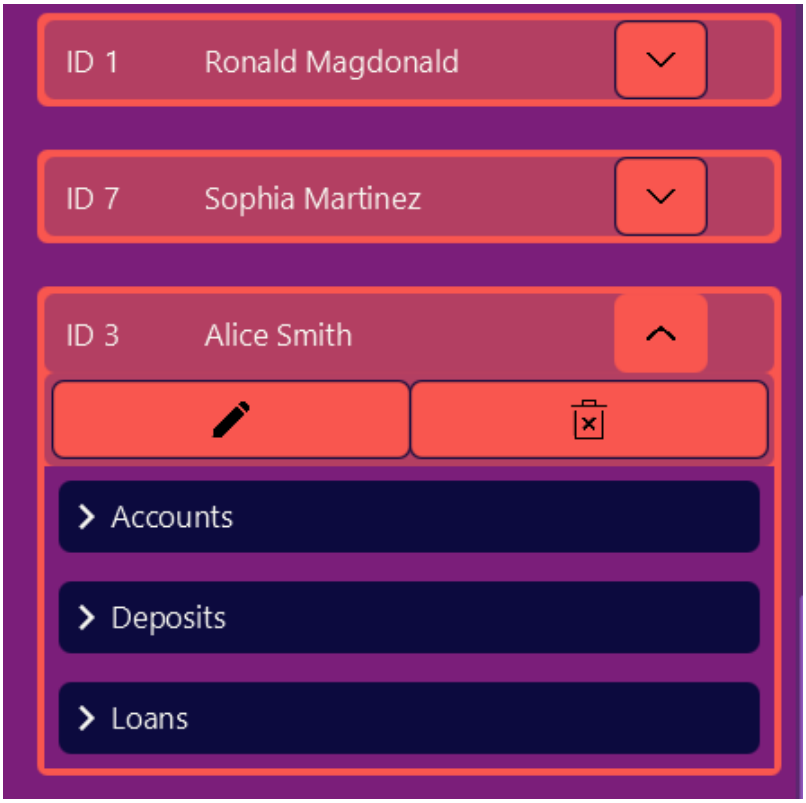
- **Lista walut**: Wyświetla wszystkie dostępne waluty.
- **Dodawanie waluty**: Przycisk otwierający formularz do dodania nowej waluty.
- **Edytowanie waluty**: Ikona edycji przy każdej walucie, otwierająca formularz do edycji wybranej waluty.
- **Usuwanie waluty**: Ikona kosza przy każdej walucie, umożliwiająca usunięcie waluty po potwierdzeniu.

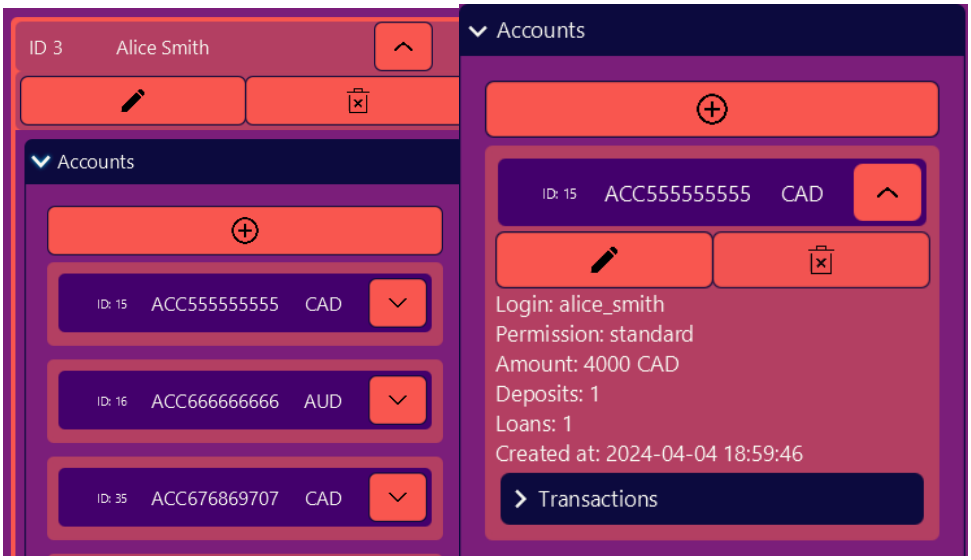


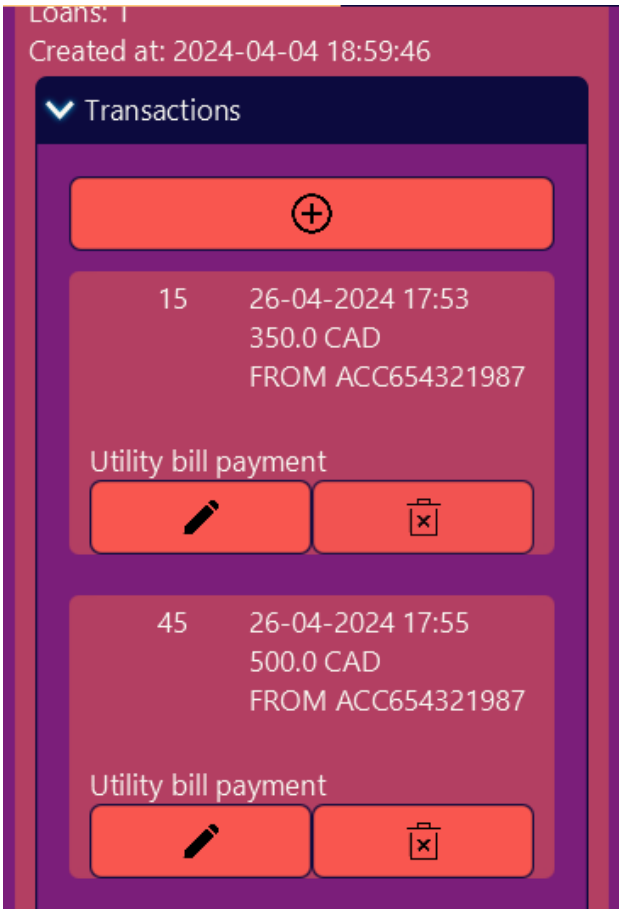
3. Panel z Użytkownikami

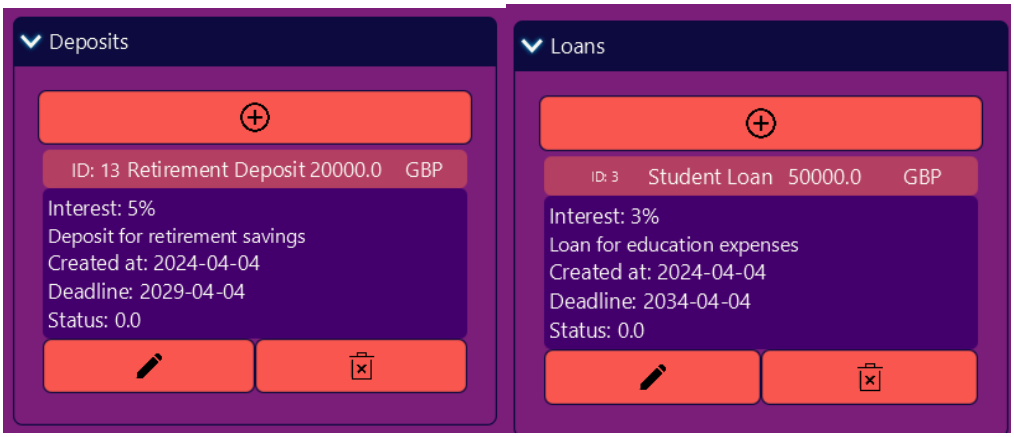
- **Lista użytkowników**: Każdy użytkownik ma swój własny panel, który można wysunąć.

- **Konta użytkownika**: Po wysunięciu panelu użytkownika, pojawia się lista kont użytkownika.
    - **Informacje o koncie**: Każde konto ma swój panel z możliwością wysunięcia szczegółowych informacji oraz transakcji.
        - **Transakcje konta**: Lista transakcji powiązanych z danym kontem.
- **Lokaty użytkownika**: Lista lokat użytkownika z możliwością wysunięcia szczegółowych informacji.
- **Pożyczki użytkownika**: Lista pożyczek użytkownika z możliwością wysunięcia szczegółowych informacji.




ID 3 Alice Smith
Accounts
ID: 15 ACC555555555 CAD
ID: 16 ACC666666666 AUD
ID: 35 ACC676869707 CAD
Accounts
ID: 15 ACC555555555 CAD
Login: alice_smith
Permission: standard
Amount: 4000 CAD
Deposits: 1
Loans: 1
Created at: 2024-04-04 18:59:46
Transactions


Loans: 1
Created at: 2024-04-04 18:59:46
Transactions
15 26-04-2024 17:53
350.0 CAD
FROM ACC654321987
Utility bill payment
45 26-04-2024 17:55
500.0 CAD
FROM ACC654321987
Utility bill payment

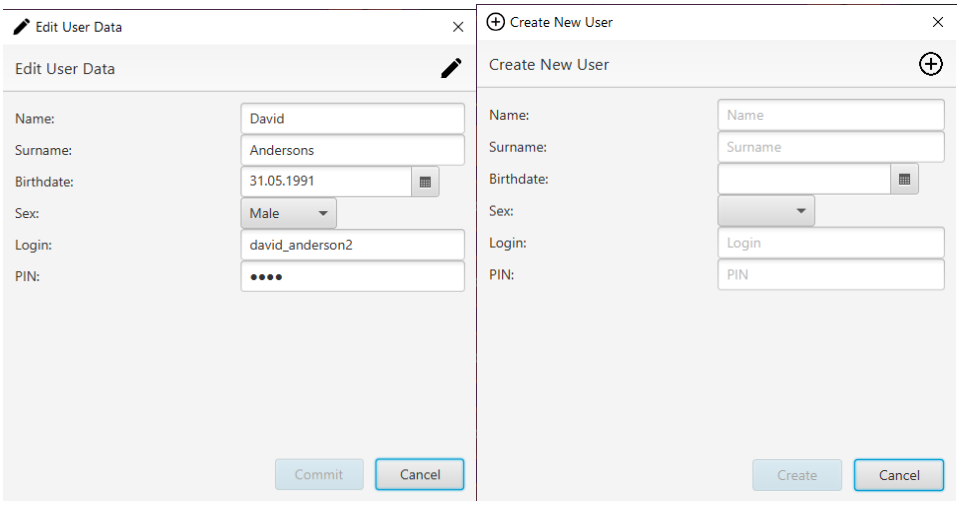


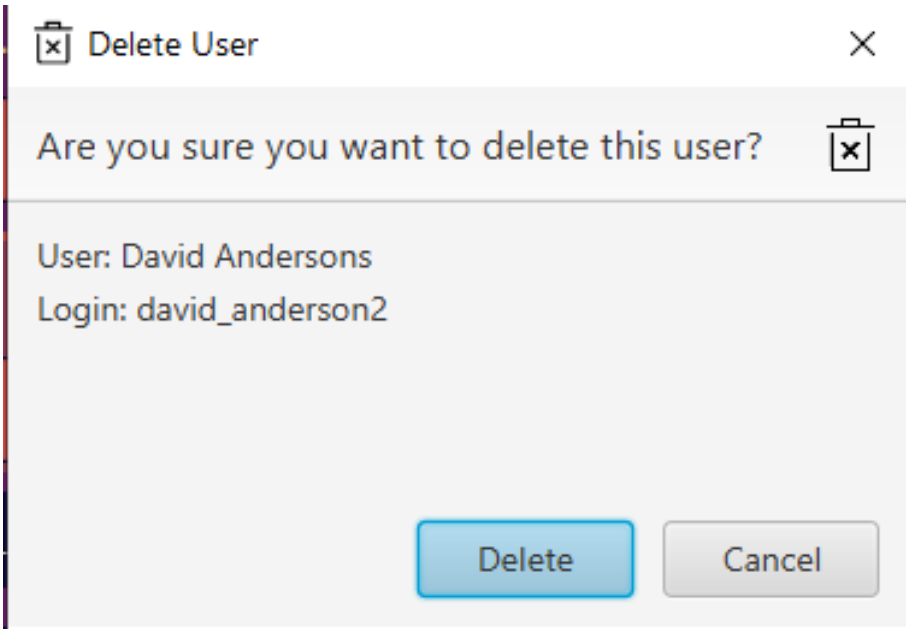
4. Operacje CRUD

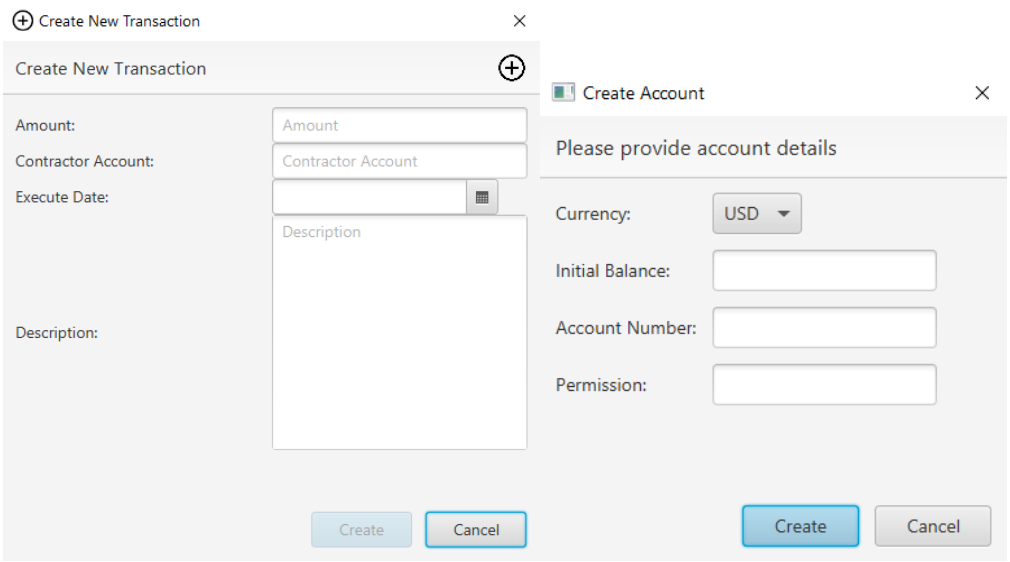
- **Dodawanie rekordów**: Przyciski otwierające formularze do dodawania nowych rekordów w odpowiednich panelach (waluty, użytkownicy, konta, lokaty, pożyczki, transakcje).
- **Edytowanie rekordów**: Ikony edycji przy każdym rekordzie, otwierające formularze do edycji.
- **Usuwanie rekordów**: Ikony kosza przy każdym rekordzie, umożliwiające usunięcie po potwierdzeniu.

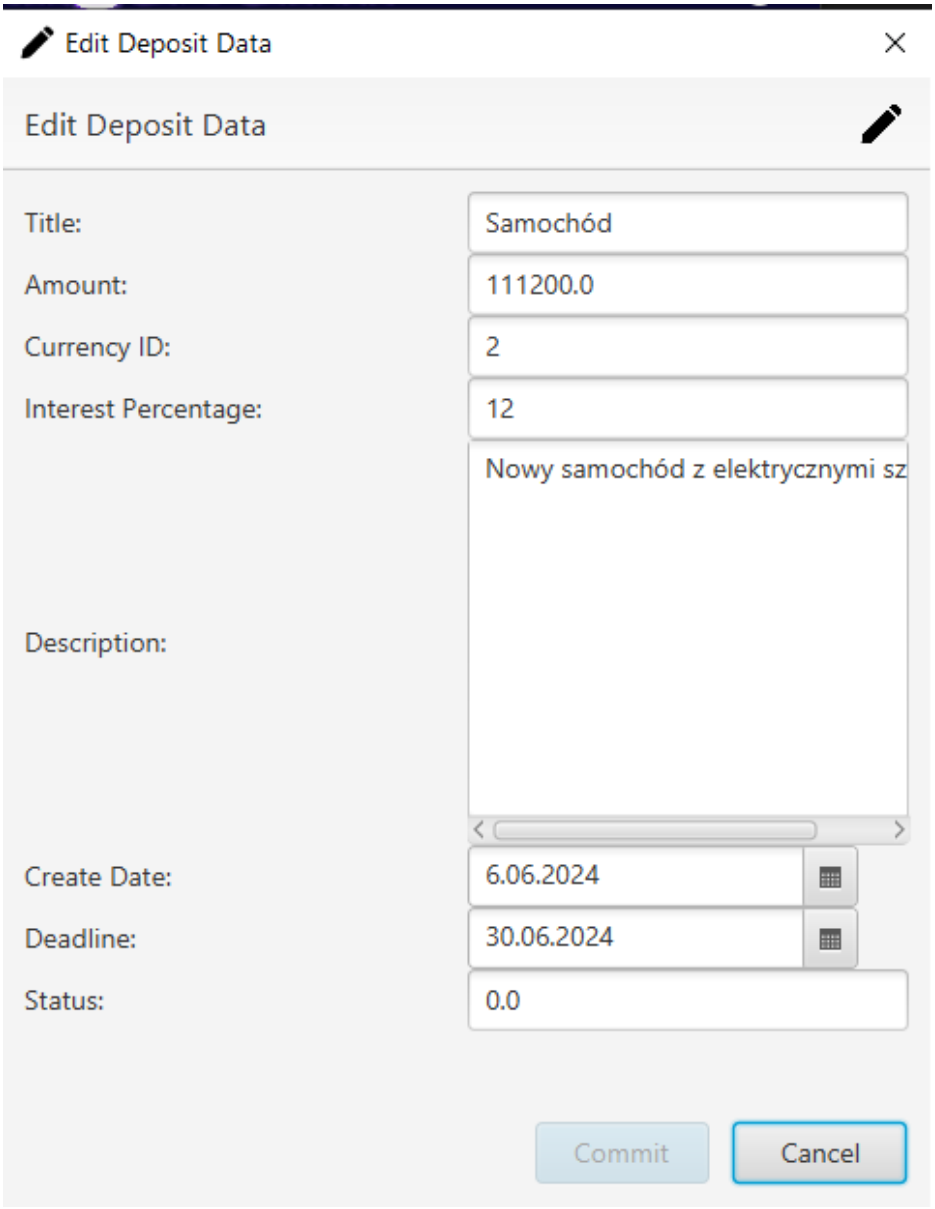
5. Formularze i Walidacja

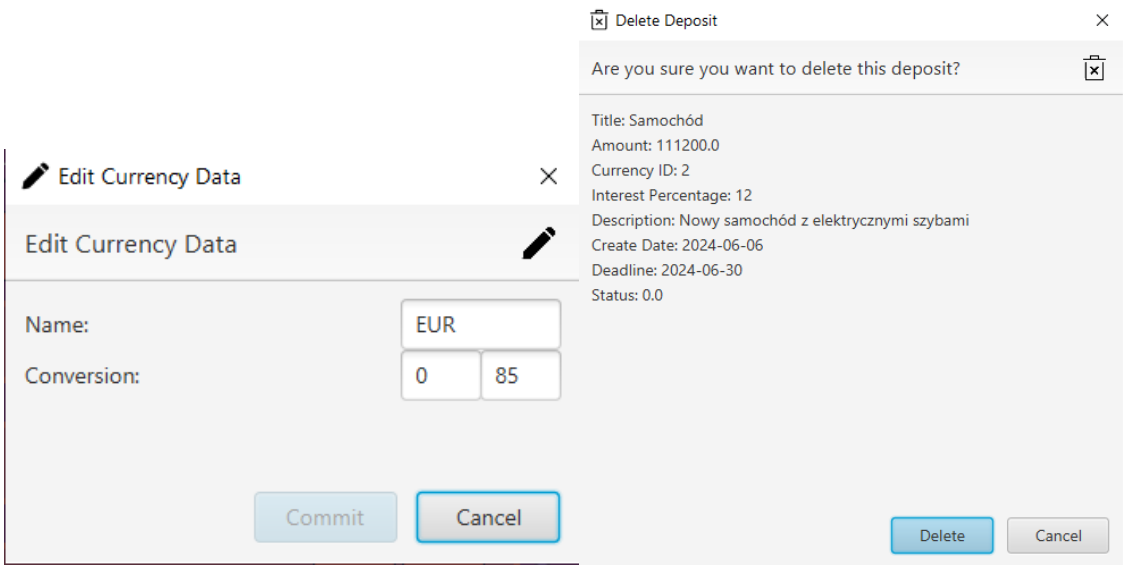
- **Formularze CRUD**: Formularze są intuicyjne, umożliwiające wygodne wprowadzanie danych.
    - **Walidacja danych**: Każde pole formularza podlega walidacji (np. sprawdzanie, czy kwoty są nieujemne, czy pola nie są puste).
    - **Komunikaty o błędach**: W przypadku niepowodzenia walidacji lub błędów serwera SQL, wyświetlane są szczegółowe komunikaty o błędach.


Edit User Data
Edit User Data
Name:
David
Surname:
Andersons
Birthdate:
31.05.1991
Sex:
Male
Login:
david_anderson2
PIN:
Commit
Cancel
Create New User
Create New User
Name:
Name
Surname:
Surname
Birthdate:
Sex:
Login:
Login
PIN:
PIN
Create
Cancel


Delete User
Are you sure you want to delete this user?
User: David Andersons
Login: david_anderson2
Delete
Cancel

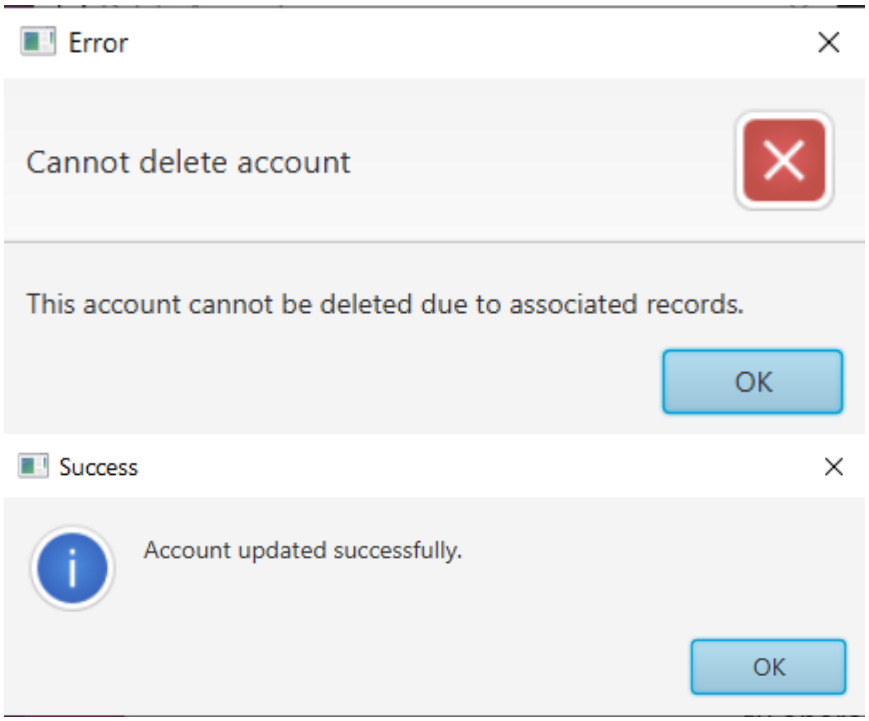

Create New Transaction
Create New Transaction
Amount:
Amount
Contractor Account:
Contractor Account
Execute Date:
Description
Description:
Create
Cancel
Create Account
Please provide account details
Currency:
USD
Initial Balance:
Account Number:
Permission:
Create
Cancel

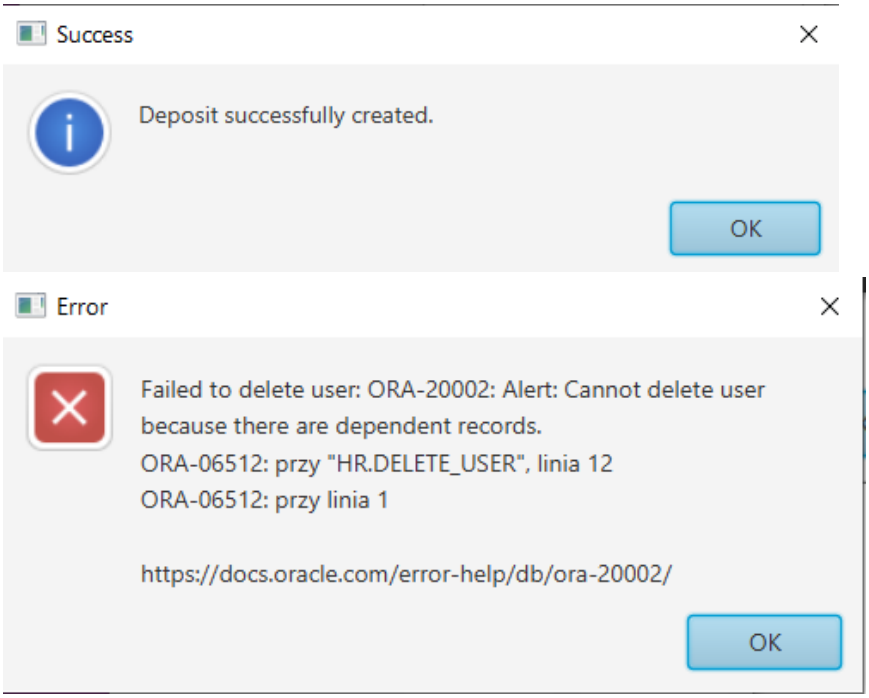

Edit Deposit Data
Edit Deposit Data
Title:
Samochód
Amount:
111200.0
Currency ID:
2
Interest Percentage:
12
Nowy samochód z elektrycznymi sz
Description:
Create Date:
6.06.2024
Deadline:
30.06.2024
Status:
0.0
Commit
Cancel



6. Komunikaty dla Użytkownika

- **Powiadomienia o powodzeniu**: Po pomyślnym wykonaniu operacji (dodanie, edytowanie, usuwanie) użytkownik otrzymuje powiadomienie.
- **Ostrzeżenia o błędach**: W przypadku błędów, użytkownik otrzymuje szczegółowe informacje o problemie, w tym komunikaty zwracane przez serwer SQL.





7. Estetyka i Intuicyjność

- **Kolorystyka**: Przyjazna dla oka kolorystyka, zapewniająca dobrą czytelność i komfort użytkowania.
- **Intuicyjne rozmieszczenie elementów**: Przejrzysty układ elementów, umożliwiający łatwe nawigowanie po aplikacji.

Uruchomienie Aplikacji

Wymagane Komponenty

43. **System Operacyjny**: Aplikacja jest kompatybilna z systemami operacyjnymi Windows, macOS oraz Linux.
44. **Serwer Baz Danych**: Konieczne jest posiadanie serwera baz danych Oracle w wersji X lub nowszej. Można skorzystać zarówno z lokalnego serwera Oracle, jak i chmurowych rozwiązań oferowanych przez dostawców.
45. **Oprogramowanie Programistyczne**: Do uruchomienia aplikacji potrzebne są środowiska programistyczne obsługujące język PL/SQL, takie jak Oracle SQL Developer lub SQL*Plus.
46. **Java Development Kit (JDK)**: Aplikacja wymaga zainstalowanego JDK w wersji 8 lub nowszej, ze względu na wykorzystanie narzędzi programistycznych oraz uruchomienie serwera baz danych Oracle.

Zależności Oprogramowania

47. **JavaFX**: Aplikacja została zbudowana przy użyciu JavaFX, więc konieczne jest jego zainstalowanie oraz konfiguracja w środowisku programistycznym.
48. **Biblioteki Maven**: Projekt może korzystać z systemu zarządzania zależnościami Maven do zarządzania bibliotekami i zależnościami. Należy upewnić się, że wszystkie wymagane biblioteki są zdefiniowane poprawnie w pliku `pom.xml`.
49. **Sterowniki JDBC**: Aplikacja łączy się z bazą danych za pomocą sterowników JDBC. Należy upewnić się, że odpowiedni sterownik JDBC dla bazy danych Oracle jest dostępny w projekcie i skonfigurowany poprawnie.
50. **Baza Danych Oracle**: Serwer baz danych Oracle musi być zainstalowany i skonfigurowany przed uruchomieniem aplikacji. Należy również utworzyć schemat bazy danych i zaimportować odpowiednie skrypty SQL.
51. **Oracle SQL Developer**: Opcjonalne, ale zalecane narzędzie do zarządzania bazą danych, umożliwiające wygodne wykonywanie zapytań, importowanie i eksportowanie danych oraz monitorowanie wydajności.

Implementacja Komponentów SQL

52. **Tworzenie Tabel**: Należy wykonać skrypt SQL zawierający polecenia `CREATE TABLE` dla wszystkich tabel używanych przez aplikację, włącznie z odpowiednimi ograniczeniami kluczy obcych i unikalności.
53. **Tworzenie Sekwencji**: Sekwencje są używane do generowania unikalnych identyfikatorów dla rekordów w tabelach. Należy utworzyć sekwencje za pomocą polecenia `CREATE SEQUENCE`.
54. **Tworzenie Wyzwalaczy (Triggerów)**: Wyzwalacze są używane do automatyzacji działań w bazie danych, na przykład do generowania wartości dla identyfikatorów przed wstawieniem rekordów. Należy utworzyć wyzwalacze za pomocą polecenia `CREATE TRIGGER`.
55. **Tworzenie Typów Obiektowych**: Typy obiektowe mogą być używane do definiowania bardziej złożonych struktur danych. Należy utworzyć typy obiektowe za pomocą polecenia `CREATE TYPE`.
56. **Tworzenie Procedur**: Procedury przechowywane są używane do wykonywania operacji na danych w bazie danych. Należy utworzyć procedury za pomocą polecenia `CREATE PROCEDURE`.

Konfiguracja Środowiska Programistycznego

57. **Połączenie z Bazą Danych**: W narzędziu programistycznym należy skonfigurować połączenie z bazą danych, dostarczając odpowiednie dane uwierzytelniające oraz informacje o serwerze.

58. **Importowanie Skryptów SQL**: Skrypty SQL zawierające definicje tabel, sekwencji, wyzwalaczy, typów obiektowych oraz procedur muszą zostać zaimportowane do środowiska programistycznego i wykonane na serwerze baz danych.
59. **Testowanie Połączenia**: Po skonfigurowaniu połączenia z bazą danych należy przetestować je, aby upewnić się, że aplikacja może komunikować się z serwerem poprawnie.

Uruchomienie Aplikacji

60. **Kompilacja i Budowanie**: Aplikacja musi zostać skompilowana i zbudowana zgodnie z wymaganiami środowiska programistycznego i narzędzi budowania projektu.
61. **Uruchomienie Serwera Baz Danych**: Przed uruchomieniem aplikacji należy upewnić się, że serwer baz danych Oracle jest uruchomiony i działa poprawnie.
62. **Uruchomienie Aplikacji**: Po spełnieniu wszystkich wymagań i skonfigurowaniu środowiska można uruchomić aplikację, korzystając z plików wykonywalnych lub za pomocą środowiska deweloperskiego.

Metody i funkcje w kodzie

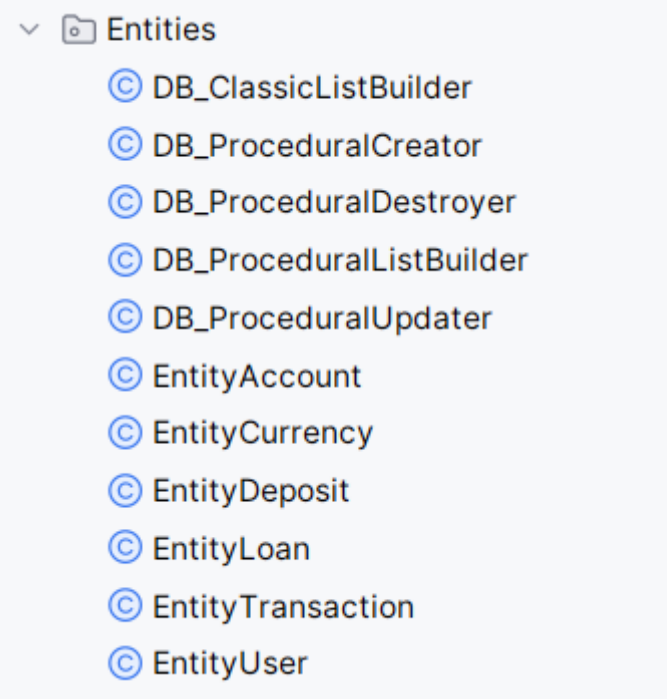
Opis Klas Mapujących

Klasy `Entity` są kluczowym elementem struktury aplikacji bazodanowej. Każda z tych klas reprezentuje jedną tabelę w bazie danych i zawiera pola odpowiadające kolumnom tabeli oraz metody dostępowe do tych pól. W praktyce, klasy te służą do przechowywania danych o poszczególnych rekordach w bazie danych i ułatwiają operacje na tych danych.

Na przykład, jeśli mamy tabelę `Users` w bazie danych, to odpowiadająca jej klasa `EntityUser` będzie zawierała pola takie jak `user_id`, `name`, `surname`, `birthdate`, `sex`, `login`, `pin`, zgodnie z kolumnami tej tabeli. Dodatkowo, klasa `EntityUser` może mieć metody dostępowe, które umożliwiają odczyt i ustawianie wartości tych pól, oraz inne metody pomocnicze.

Klasy `Entity` są wykorzystywane w wielu miejscach aplikacji, między innymi do wykonywania operacji CRUD (Create, Read, Update, Delete) na danych w bazie danych, do tworzenia obiektów reprezentujących te dane w warstwie logiki biznesowej aplikacji, oraz do przesyłania danych między różnymi komponentami aplikacji. Dzięki

zastosowaniu tych klas, struktura danych w aplikacji jest jasna i zorganizowana, co ułatwia zarządzanie nimi i rozwijanie aplikacji.




```java
public class EntityUser {  dyjak

    int user_id;  4 usages
    String name;  4 usages
    String surname;  4 usages
    Date birthdate;  4 usages
    String sex;  4 usages
    String login;  4 usages
    String pin;  4 usages

    public EntityUser(int user_id, String name, String surname, Date birthdate, String sex, String login, String pin) {
        this.user_id = user_id;
        this.name = name;
        this.surname = surname;
        this.birthdate = birthdate;
        this.sex = sex;
        this.login = login;
        this.pin = pin;
    }

    public int getUser_id() { return user_id; }

    public void setUser_id(int user_id) { this.user_id = user_id; }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getSurname() { return surname; }

    public void setSurname(String surname) { this.surname = surname; }
```

Opis Funkcji Klas Budujących Listę Obiektów Tabeli

Klasa: DB_ProceduralListBuilder

63. **Metoda: `userListBuild(String argument1, String argument2)`**
    - Opis: Metoda ta służy do budowania listy użytkowników na podstawie wyników procedury składowanej `search_users`.
    - Parametry:
        - `argument1`: Pierwszy argument przekazywany do procedury `search_users`.
        - `argument2`: Drugi argument przekazywany do procedury `search_users`.
    - Zwracany Wynik: Lista obiektów klasy `EntityUser`.
64. **Metoda: `currencyListBuild(String argument)`**
    - Opis: Metoda ta służy do budowania listy walut na podstawie wyników procedury składowanej `search_currencies`.
    - Parametry:
        - `argument`: Argument przekazywany do procedury `search_currencies`.
    - Zwracany Wynik: Lista obiektów klasy `EntityCurrency`.
65. **Metoda: `accountInfoBuild(int argument)`**
    - Opis: Metoda ta służy do pobierania informacji o konkretnym koncie na podstawie jego identyfikatora.
    - Parametry:
        - `argument`: Identyfikator konta.
    - Zwracany Wynik: Tablica `String[]` zawierająca informacje o koncie.

Klasa: DB_ClassicListBuilder

66. **Metoda: `accountListBuild(String sqlQuery)`**
    - Opis: Metoda ta służy do budowania listy kont na podstawie zapytania SQL.
    - Parametry:
        - `sqlQuery`: Zapytanie SQL zwracające wyniki.
    - Zwracany Wynik: Lista obiektów klasy `EntityAccount`.
67. **Metoda: `currencyBuild(String sqlQuery)`**
    - Opis: Metoda ta służy do budowania pojedynczego obiektu waluty na podstawie zapytania SQL.
    - Parametry:
        - `sqlQuery`: Zapytanie SQL zwracające wynik pojedynczej waluty.

- Zwracany Wynik: Obiekt klasy `EntityCurrency`.

68. **Metoda: `allCurrenciesBuild()`**
    - Opis: Metoda ta służy do budowania listy wszystkich dostępnych walut.
    - Zwracany Wynik: Lista obiektów klasy `EntityCurrency`.
69. **Metoda: `depositListBuild(String sqlQuery)`**
    - Opis: Metoda ta służy do budowania listy lokat na podstawie zapytania SQL.
    - Parametry:
        - `sqlQuery`: Zapytanie SQL zwracające wyniki.
    - Zwracany Wynik: Lista obiektów klasy `EntityDeposit`.
70. **Metoda: `loanListBuild(String sqlQuery)`**
    - Opis: Metoda ta służy do budowania listy pożyczek na podstawie zapytania SQL.
    - Parametry:
        - `sqlQuery`: Zapytanie SQL zwracające wyniki.
    - Zwracany Wynik: Lista obiektów klasy `EntityLoan`.
71. **Metoda: `transactionListBuild(String sqlQuery)`**
    - Opis: Metoda ta służy do budowania listy transakcji na podstawie zapytania SQL.
    - Parametry:
        - `sqlQuery`: Zapytanie SQL zwracające wyniki.
    - Zwracany Wynik: Lista obiektów klasy `EntityTransaction`.
72. **Metoda: `userListBuild(String sqlQuery)`**
    - Opis: Metoda ta służy do budowania listy użytkowników na podstawie zapytania SQL.
    - Parametry:
        - `sqlQuery`: Zapytanie SQL zwracające wyniki.
    - Zwracany Wynik: Lista obiektów klasy `EntityUser`.

```java
//USERS BUILDER
public ArrayList<EntityUser> userListBuild(String argument1, String argument2) throws SQLException {  1 usage  dyjak
    Statement createStatement = DB_EstablishConnection().createStatement();

    CallableStatement callableStatement = DB_EstablishConnection().prepareCall( sql: "{CALL search_users(?, ?)}");
    callableStatement.setString( parameterIndex: 1, argument1);
    callableStatement.setString( parameterIndex: 2, argument2);
    callableStatement.execute();

    String sqlQuery="SELECT * FROM temp_users";
    ResultSet resultSet = DB_EstablishConnection().createStatement().executeQuery(sqlQuery);

    ArrayList<EntityUser> users_x = new ArrayList<>();
    while (resultSet.next()) {
        int user_id = resultSet.getInt( columnLabel: "user_id");
        String name = resultSet.getString( columnLabel: "name");
        String surname = resultSet.getString( columnLabel: "surname");
        Date birthdate = resultSet.getDate( columnLabel: "birthdate");
        String sex = resultSet.getString( columnLabel: "sex");
        String login = resultSet.getString( columnLabel: "login");
        String pin = resultSet.getString( columnLabel: "pin");

        EntityUser user = new EntityUser(user_id, name, surname, birthdate, sex, login, pin);
        users_x.add(user);
    }
    return users_x;
}
```

Opis Funkcji Klas Obsługujących Operacje CRUD

Klasa: DB_ProceduralCreator

73. **Metoda: `currencyCreate(String argName, double argConversion)`**
    - Opis: Metoda ta służy do dodawania nowej waluty do bazy danych.
    - Parametry:
        - `argName`: Nazwa waluty.
        - `argConversion`: Współczynnik konwersji na USD.
    - Wynik: Metoda wyświetla komunikat "SUCCESS!" po pomyślnym dodaniu.
74. **Metoda: `userCreate(String name, String surname, String birthDate, String sex, String login, String pin)`**
    - Opis: Metoda ta służy do dodawania nowego użytkownika do bazy danych.
    - Parametry:
        - `name`: Imię użytkownika.
        - `surname`: Nazwisko użytkownika.
        - `birthDate`: Data urodzenia w formacie "YYYY-MM-DD".
        - `sex`: Płeć użytkownika.
        - `login`: Nazwa użytkownika do logowania.
        - `pin`: PIN użytkownika.
    - Wynik: Metoda wyświetla komunikat "SUCCESS!" po pomyślnym dodaniu.
75. **Metoda: `accountCreate(int p_user_id, int p_currency_id, double p_balance, String p_account_number, String p_permission)`**
    - Opis: Metoda ta służy do dodawania nowego konta do bazy danych.

- Parametry:
  - `p_user_id`: Identyfikator użytkownika przypisanego do konta.
  - `p_currency_id`: Identyfikator waluty przypisanej do konta.
  - `p_balance`: Saldo konta.
  - `p_account_number`: Numer konta.
  - `p_permission`: Uprawnienia konta.
- Wynik: Metoda wyświetla komunikat "SUCCESS!" po pomyślnym dodaniu.

76. **Metoda: `transactionCreate(int accountId, double amount, String contractorAccountNumber, Timestamp executeDate, String description)`**
    - Opis: Metoda ta służy do dodawania nowej transakcji do bazy danych.
    - Parametry:
      - `accountId`: Identyfikator konta, na którym ma być wykonana transakcja.
      - `amount`: Kwota transakcji.
      - `contractorAccountNumber`: Numer konta kontrahenta.
      - `executeDate`: Data wykonania transakcji.
      - `description`: Opis transakcji.
    - Wynik: Metoda wyświetla komunikat "SUCCESS!" po pomyślnym dodaniu.
77. **Metoda: `depositCreate(String title, double amount, int currencyId, double interestPercentage, String description, int userId, Date createDate, Date deadline, double status)`**
    - Opis: Metoda ta służy do dodawania nowej lokaty do bazy danych.
    - Parametry:
      - `title`: Tytuł lokaty.
      - `amount`: Kwota lokaty.
      - `currencyId`: Identyfikator waluty lokaty.
      - `interestPercentage`: Oprocentowanie lokaty.
      - `description`: Opis lokaty.
      - `userId`: Identyfikator użytkownika przypisanego do lokaty.
      - `createDate`: Data utworzenia lokaty.
      - `deadline`: Termin wykonania lokaty.
      - `status`: Status lokaty.
    - Wynik: Metoda wyświetla komunikat "Deposit added successfully!" po pomyślnym dodaniu.
78. **Metoda: `loanCreate(String title, double amount, int currencyId, double interestPercentage, String description, int userId, Date createDate, Date deadline, double status)`**
    - Opis: Metoda ta służy do dodawania nowej pożyczki do bazy danych.
    - Parametry:

- `title`: Tytuł pożyczki.
- `amount`: Kwota pożyczki.
- `currencyId`: Identyfikator waluty pożyczki.
- `interestPercentage`: Oprocentowanie pożyczki.
- `description`: Opis pożyczki.
- `userId`: Identyfikator użytkownika przypisanego do pożyczki.
- `createDate`: Data utworzenia pożyczki.
- `deadline`: Termin wykonania pożyczki.
- `status`: Status pożyczki.

- Wynik: Metoda wyświetla komunikat "Loan added successfully!" po pomyślnym dodaniu.

Klasa: DB_ProceduralDestroyer

79. Metoda: `currencyDestroy(int argID)`

- Opis: Metoda ta służy do usuwania waluty z bazy danych na podstawie jej identyfikatora.
- Parametry:
  - `argID`: Identyfikator waluty do usunięcia.
- Wynik: Metoda wyświetla komunikat "SUCCESS!" po pomyślnym usunięciu.

80. Metoda: `userDestroy(int argID)`

- Opis: Metoda ta służy do usuwania użytkownika z bazy danych na podstawie jego identyfikatora.
- Parametry:
  - `argID`: Identyfikator użytkownika do usunięcia.
- Wynik: Metoda wyświetla komunikat "SUCCESS!" po pomyślnym usunięciu.

81. Metoda: `accountDestroy(int accountID)`

- Opis: Metoda ta służy do usuwania konta z bazy danych na podstawie jego identyfikatora.
- Parametry:
  - `accountID`: Identyfikator konta do usunięcia.
- Wynik: Metoda usuwa konto z bazy danych.

82. Metoda: `transactionDestroy(int transactionId)`

- Opis: Metoda ta służy do usuwania transakcji z bazy danych na podstawie jej identyfikatora.
- Parametry:
  - `transactionId`: Identyfikator transakcji do usunięcia.

- Wynik: Metoda wyświetla komunikat "SUCCESS!" po pomyślnym usunięciu.

83. **Metoda: `deleteDeposit(int depositId)`**
    - Opis: Metoda ta służy do usuwania lokaty z bazy danych na podstawie jej identyfikatora.
- Parametry:
    - `depositId`: Identyfikator lokaty do usunięcia.
- Wynik: Metoda wyświetla komunikat "Deposit deleted successfully!" po pomyślnym usunięciu.
- **Metoda: `deleteLoan(int loanId)`**
    - Opis: Metoda ta służy do usuwania pożyczki z bazy danych na podstawie jej identyfikatora.
    - Parametry:
        - `loanId`: Identyfikator pożyczki do usunięcia.
    - Wynik: Metoda wyświetla komunikat "Loan deleted successfully!" po pomyślnym usunięciu.

- ***Klasa: `DB_ProceduralUpdater`***

- **Metoda: `currencyUpdate(String argName, double argConversion, int argID)`**
    - Opis: Metoda ta służy do aktualizowania danych waluty w bazie danych na podstawie jej identyfikatora.
    - Parametry:
        - `argName`: Nowa nazwa waluty.
        - `argConversion`: Nowy współczynnik konwersji na USD.
        - `argID`: Identyfikator waluty do zaktualizowania.
    - Wynik: Metoda wyświetla komunikat "SUCCESS!" po pomyślnym zaktualizowaniu.
- **Metoda: `userUpdate(int userId, String name, String surname, java.sql.Date birthDate, String sex, String login, String pin)`**
    - Opis: Metoda ta służy do aktualizowania danych użytkownika w bazie danych na podstawie jego identyfikatora.
    - Parametry:
        - `userId`: Identyfikator użytkownika do zaktualizowania.
        - `name`: Nowe imię użytkownika.
        - `surname`: Nowe nazwisko użytkownika.
        - `birthDate`: Nowa data urodzenia użytkownika.
        - `sex`: Nowa płeć użytkownika.
        - `login`: Nowa nazwa użytkownika do logowania.

        - `pin`: Nowy PIN użytkownika.
    - Wynik: Metoda wyświetla komunikat "User updated successfully!" po pomyślnym zaktualizowaniu.
- **Metoda: `accountUpdate(EntityAccount account)`**
    - Opis: Metoda ta służy do aktualizowania danych konta w bazie danych.
    - Parametry:
        - `account`: Obiekt klasy `EntityAccount` zawierający zaktualizowane dane konta.
    - Wynik: Metoda wyświetla komunikat "SUCCESS!" po pomyślnym zaktualizowaniu.
- **Metoda: `transactionUpdate(int transactionId, int accountId, double amount, String contractorAccountNumber, String description)`**
    - Opis: Metoda ta służy do aktualizowania danych transakcji w bazie danych na podstawie jej identyfikatora.
    - Parametry:
        - `transactionId`: Identyfikator transakcji do zaktualizowania.
        - `accountId`: Identyfikator konta, na którym ma być wykonana transakcja.
        - `amount`: Nowa kwota transakcji.
        - `contractorAccountNumber`: Nowy numer konta kontrahenta.
        - `description`: Nowy opis transakcji.
    - Wynik: Metoda wyświetla komunikat "Transaction updated successfully!" po pomyślnym zaktualizowaniu.
- **Metoda: `depositUpdate(int depositId, String title, double amount, int currencyId, double interestPercentage, String description, Date createDate, Date deadline, double status)`**
    - Opis: Metoda ta służy do aktualizowania danych lokaty w bazie danych na podstawie jej identyfikatora.
    - Parametry:
        - `depositId`: Identyfikator lokaty do zaktualizowania.
        - `title`: Nowy tytuł lokaty.
        - `amount`: Nowa kwota lokaty.
        - `currencyId`: Nowy identyfikator waluty lokaty.
        - `interestPercentage`: Nowe oprocentowanie lokaty.
        - `description`: Nowy opis lokaty.
        - `createDate`: Nowa data utworzenia lokaty.
        - `deadline`: Nowy termin wykonania lokaty.
        - `status`: Nowy status lokaty.

    - Wynik: Metoda wyświetla komunikat "Deposit updated successfully!" po pomyślnym zaktualizowaniu.

- **Metoda: `loanUpdate(int loanId, String title, double amount, int currencyId, double interestPercentage, String description, Date createDate, Date deadline, double status)`**
- Opis: Metoda ta służy do aktualizowania danych pożyczki w bazie danych na podstawie jej identyfikatora.
- Parametry:
    - `loanId`: Identyfikator pożyczki do zaktualizowania.
    - `title`: Nowy tytuł pożyczki.
    - `amount`: Nowa kwota pożyczki.
    - `currencyId`: Nowy identyfikator waluty pożyczki.
    - `interestPercentage`: Nowe oprocentowanie pożyczki.
    - `description`: Nowy opis pożyczki.
    - `createDate`: Nowa data utworzenia pożyczki.
    - `deadline`: Nowy termin wykonania pożyczki.
    - `status`: Nowy status pożyczki.
- Wynik: Metoda wyświetla komunikat "Loan updated successfully!" po pomyślnym zaktualizowaniu.

```java
//CONNECTION ESTABLISHMENT
private static Connection DB_EstablishConnection() throws SQLException {  9 usages  dyjak
    String jdbcURL = "jdbc:oracle:thin:@//localhost:1521/freepdb1";
    String username = "hr";
    String password = "oracle";
    Connection connection = DriverManager.getConnection(jdbcURL, username, password);
    return connection;
}

//CREATE CURRENCY
public void currencyCreate(String argName, double argConversion) throws SQLException {  1 usage  dyjak
    Statement createStatement = DB_EstablishConnection().createStatement();
    CallableStatement callableStatement = DB_EstablishConnection().prepareCall( sql: "{CALL add_currency(?, ?)}");
    callableStatement.setString( parameterIndex: 1, argName);
    callableStatement.setDouble( parameterIndex: 2, argConversion);
    callableStatement.execute();
    callableStatement.close();
    System.out.println("SUCCESS!");
}
```

```java
//DESTROY ACCOUNT
public void accountDestroy(int accountID) throws SQLException {  1 usage  dyjak
    try (Connection connection = DB_EstablishConnection();
         CallableStatement callableStatement = connection.prepareCall( sql: "{CALL delete_account(?)}")) {
        callableStatement.setInt( parameterIndex: 1, accountID);
        callableStatement.execute();
    }
}

//DESTROY TRANSACTION
public void transactionDestroy(int transactionId) throws SQLException {  1 usage  dyjak
    CallableStatement callableStatement = DB_EstablishConnection().prepareCall( sql: "{CALL delete_transaction(?)}");
    callableStatement.setInt( parameterIndex: 1, transactionId);
    callableStatement.execute();
    callableStatement.close();
    System.out.println("SUCCESS!");
}
```

```java
//UPDATE CURRENCY
public void currencyUpdate(String argName, double argConversion, int argID) throws SQLException {  1 usage  dyjak
    Statement createStatement = DB_EstablishConnection().createStatement();
    CallableStatement callableStatement = DB_EstablishConnection().prepareCall( sql: "{CALL update_currency(?, ?, ?)}");
    callableStatement.setString( parameterIndex: 1, argName);
    callableStatement.setDouble( parameterIndex: 2, argConversion);
    callableStatement.setInt( parameterIndex: 3, argID);
    callableStatement.execute();
    callableStatement.close();
    System.out.println("SUCCESS!");
}
```

Wykorzystanie prefabrykatów podczas pisania programu

Prefabrykaty (ang. prefabs) są to wzorce lub szablony obiektów, które można wielokrotnie wykorzystywać w aplikacji. W kontekście programowania, prefabrykaty są szczególnie użyteczne w systemach, gdzie wiele instancji obiektów ma podobne lub identyczne właściwości i zachowania.

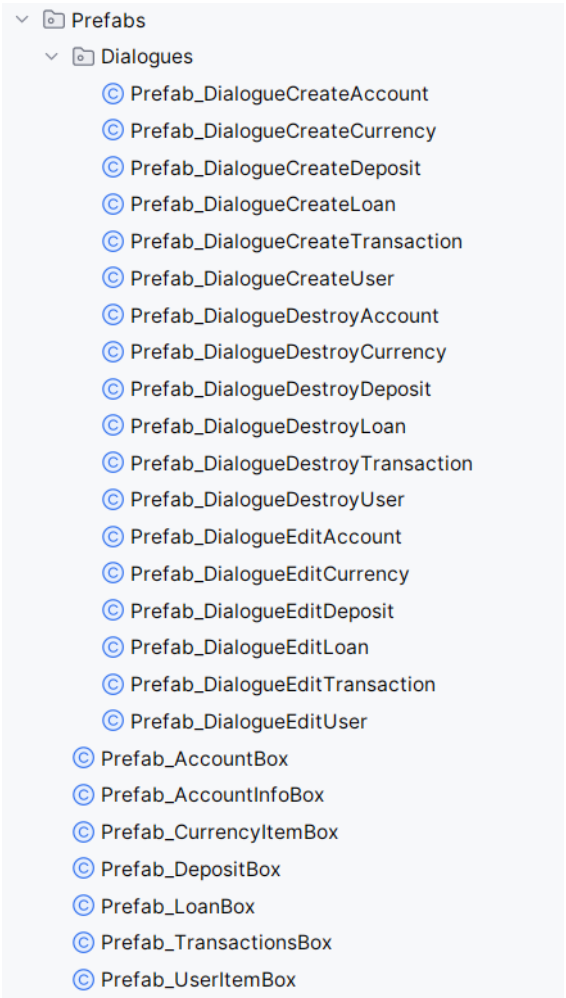
W przypadku tworzenia aplikacji, takiej jak system bankowy, prefabrykaty mogą być używane do zdefiniowania standardowych komponentów interfejsu użytkownika, jak również do definiowania standardowych operacji na danych. Na przykład, w systemie bankowym można stworzyć prefabrykaty dla różnych formularzy dodawania, edycji i usuwania danych użytkowników, kont, pożyczek, lokat itp. Te prefabrykaty mogą zawierać już zaimplementowane logiki biznesowe oraz interfejsy użytkownika, co pozwala na szybsze tworzenie nowych funkcjonalności poprzez ponowne wykorzystanie istniejących komponentów.

Korzyści z korzystania z prefabrykatów obejmują:

84. **Skrócenie czasu tworzenia aplikacji**: Ponowne wykorzystanie istniejących prefabrykatów pozwala na szybsze tworzenie nowych funkcji bez potrzeby implementowania wszystkiego od zera.
85. **Zwiększenie spójności aplikacji**: Używanie tych samych prefabrykatów w różnych częściach aplikacji pomaga utrzymać spójność interfejsu użytkownika i logiki biznesowej.
86. **Łatwiejsze zarządzanie kodem**: Dzięki zastosowaniu prefabrykatów, kod jest bardziej modułowy i łatwiejszy do zarządzania, co ułatwia późniejsze utrzymanie i rozwijanie aplikacji.
87. **Poprawa jakości**: Dzięki ponownemu użyciu sprawdzonych prefabrykatów, zmniejsza się ryzyko wprowadzenia błędów i zapewnia się wyższą jakość kodu.

Pisząc aplikację wykorzystującą prefabrykaty, ważne jest dobranie odpowiednich wzorców projektowych i architektonicznych, takich jak np. wzorzec MVC (Model-View-Controller) lub MVVM (Model-View-ViewModel), które pomogą w zachowaniu czytelności kodu, separacji warstw oraz łatwości testowania i rozwijania aplikacji.

```java
public class Prefab_LoanBox {  2 usages  dyjak

    public void show(VBox loansBox, ArrayList<EntityLoan> loans_x) {  dyjak

        for (EntityLoan loan : loans_x) {
            VBox expandedBox = new VBox();
            VBox moreItemsBox = new VBox();
            moreItemsBox.setPrefWidth(999);
            moreItemsBox.getStyleClass().add("otherBox");

            Text loanIdText = new Text("   ID: " + loan.getLoan_id());
            loanIdText.setFont(new Font( size: 9));
            Text loanTitleText = new Text(loan.getTitle());
            Text loanAmountText = new Text(String.valueOf(loan.getAmount()));
            Text loanCurrencyText = new Text("unknown");

            Text loanInterestText = new Text("Interest: " + loan.getInterest_percentage()+"%");
            Text loanDescriptionText = new Text(loan.getDescription());
            loanDescriptionText.setFont(new Font( size: 13));
            Text loanCreateDateText = new Text("Created at: " + loan.getCreate_date());
            Text loanDeadLineText = new Text("Deadline: " + loan.getDeadline());
            Text loanStatusText = new Text("Status: " + loan.getStatus());


            try {
                String query = "SELECT * FROM currencies WHERE currencies.currency_id = " + loan.getCurrency_id();
                System.out.println(query);
                EntityCurrency currency = DB_ClassicListBuilder.currencyBuild(query);
                loanCurrencyText.setText(currency.getCurrency());
            } catch (SQLException e) {
                throw new RuntimeException(e);
```



Podsumowanie

Podsumowując, tworzenie kompleksowych aplikacji, takich jak system bankowy, wymaga uwzględnienia wielu czynników i zagadnień. Od projektowania interfejsu użytkownika po implementację logiki biznesowej, od zarządzania danymi po bezpieczeństwo aplikacji - każdy aspekt wymaga uwagi i staranności.

W naszym projekcie, staraliśmy się zastosować dobre praktyki programistyczne oraz wykorzystać narzędzia i technologie, które umożliwią nam skuteczne i wydajne tworzenie aplikacji. Począwszy od wyboru odpowiednich języków programowania i frameworków, poprzez projektowanie bazy danych i interfejsu użytkownika, aż po implementację funkcjonalności i testowanie oprogramowania - każdy etap wymagał starannej analizy i podejmowania decyzji.

Stawiając czoła wyzwaniom związanym z bezpieczeństwem aplikacji, dbaliśmy o to, aby nasz system był odporny na ataki zewnętrzne i zapewniał poufność, integralność i dostępność danych. Wykorzystaliśmy różne mechanizmy uwierzytelniania i autoryzacji, szyfrowanie danych oraz zabezpieczenia warstwy aplikacji i bazy danych.

Tworzenie aplikacji bankowej to proces ciągły, który wymaga ciągłego doskonalenia i aktualizacji. Nasza praca nie kończy się na etapie implementacji - będziemy nadal monitorować działanie systemu, reagować na ewentualne problemy i dostosowywać się do zmieniających się wymagań i trendów technologicznych.

Ważne jest również zapewnienie odpowiedniej dokumentacji oraz szkoleń dla użytkowników aplikacji, aby zapewnić im płynne i efektywne korzystanie z systemu. Wraz z rozwojem technologicznym i dynamicznym środowiskiem biznesowym, będziemy dążyć do ciągłego doskonalenia naszej aplikacji, aby sprostać oczekiwaniom i potrzebom naszych użytkowników.