Akademia Górniczo-Hutnicza
Wydział Fizyki i Informatyki Stosowanej
Kierunek Informatyka Stosowana

Bazy Danych 2

Przetwarzanie Rozproszone w SQL Server

Dokumentacja Techniczna

Piotrowski Dawid

6 czerwca 2025

Spis treści

1 Wprowadzenie

Współczesne systemy bankowe i inne systemy transakcyjne często muszą zapewniać **atomowość i spójność** operacji finansowych wykonywanych w środowiskach rozproszonych. Przykładem problemu jest przelew środków pomiędzy dwoma oddziałami banku, posiadającymi osobne bazy danych. Bez specjalnych mechanizmów istnieje ryzyko, że środki zostaną obciążone z konta nadawcy, ale nie zostaną dopisane do konta odbiorcy (lub odwrotnie) w przypadku awarii jednego z systemów w trakcie transakcji. Aby zagwarantować, że transakcja rozproszona zostanie wykonana w całości albo wcale, stosuje się **protokół dwufazowego zatwierdzania** (2PC, ang. *Two-Phase Commit*) oraz mechanizmy koordynujące takie transakcje, np. **Microsoft Distributed Transaction Coordinator (MSDTC)**.

Projekt „Przetwarzanie Rozproszone w SQL Server" powstał z motywacji demonstracji działania transakcji rozproszonych w środowisku bazodanowym Microsoft SQL Server. Celem było stworzenie przykładowej biblioteki (API) oraz aplikacji klienckiej, które umożliwią wykonywanie rozproszonych operacji (np. przelewów bankowych) na dwóch bazach danych reprezentujących centralę banku oraz oddziały. Dzięki zastosowaniu protokołu 2PC i MSDTC, operacje finansowe obejmujące dwie (lub więcej) bazy danych zachowują **zasadę ACID** – szczególnie trwałość i atomowość – nawet w razie błędów sieci lub systemu. Dokumentacja ta przedstawia szczegóły techniczne implementacji projektu, w tym opis protokołu, struktur danych, implementację kodu (C# oraz SQL), testy oraz działanie aplikacji demonstracyjnej.

2 Opis problemu i funkcjonalności API

2.1 Transakcje rozproszone i protokół 2PC

W środowisku rozproszonym, gdzie jedna transakcja obejmuje wiele niezależnych zasobów (np. kilka baz danych na różnych serwerach), wymagane jest specjalne podejście do zatwierdzania (*commit*) zmian. **Protokół dwufazowego zatwierdzania** (2PC) spełnia to zadanie, działając w dwóch etapach. W *fazie pierwszej* koordynator transakcji wysyła do wszystkich uczestniczących baz danych zapytanie, czy są one gotowe do zatwierdzenia transakcji (*prepare*). Każdy uczestnik lokalnie wykonuje transakcję do punktu przygotowania i odpowiada koordynatorowi "gotów" lub zgłasza problem. Jeżeli *choć jeden uczestnik* zgłosi brak gotowości, koordynator przerywa transakcję. W przeciwnym razie następuje *faza druga*: koordynator rozsyła polecenie zatwierdzenia (*commit*) do wszystkich uczestników, którzy finalizują zmiany lub – w razie wcześniej wykrytego błędu – polecenie wycofania (*rollback*) transakcji we wszystkich bazach. Dzięki 2PC wszystkie węzły systemu podejmują **jednolitą decyzję** co do trwałości transakcji – albo każda baza zatwierdzi zmiany, albo żadna z nich.

W architekturze Microsoft SQL Server za koordynację transakcji rozproszonych odpowiada usługa **MSDTC (Microsoft Distributed Transaction Coordinator)**. Jest to usługa systemowa, która pełni rolę koordynatora 2PC – zarządza rozproszoną transakcją obejmującą wielu menedżerów zasobów (np. instancje SQL Server). W naszym projekcie MSDTC zapewnia, że operacje wykonywane w bazie centralnej i bazach oddziałów w ramach jednej transakcji rozproszonej zostaną zgodnie z protokołem 2PC zatwierdzone lub wycofane wspólnie. Należy zaznaczyć, że do poprawnego działania transakcji rozproszonych wymagana jest odpowiednia konfiguracja MSDTC na serwerach (np. włączenie dostępu sieciowego DTC, właściwe ustawienia zapory itp.), co jest jednak zagadnieniem środowiskowym – przyjmujemy, że środowisko bazodanowe jest skonfigurowane prawidłowo do obsługi DTC.

2.2 Funkcjonalności biblioteki API

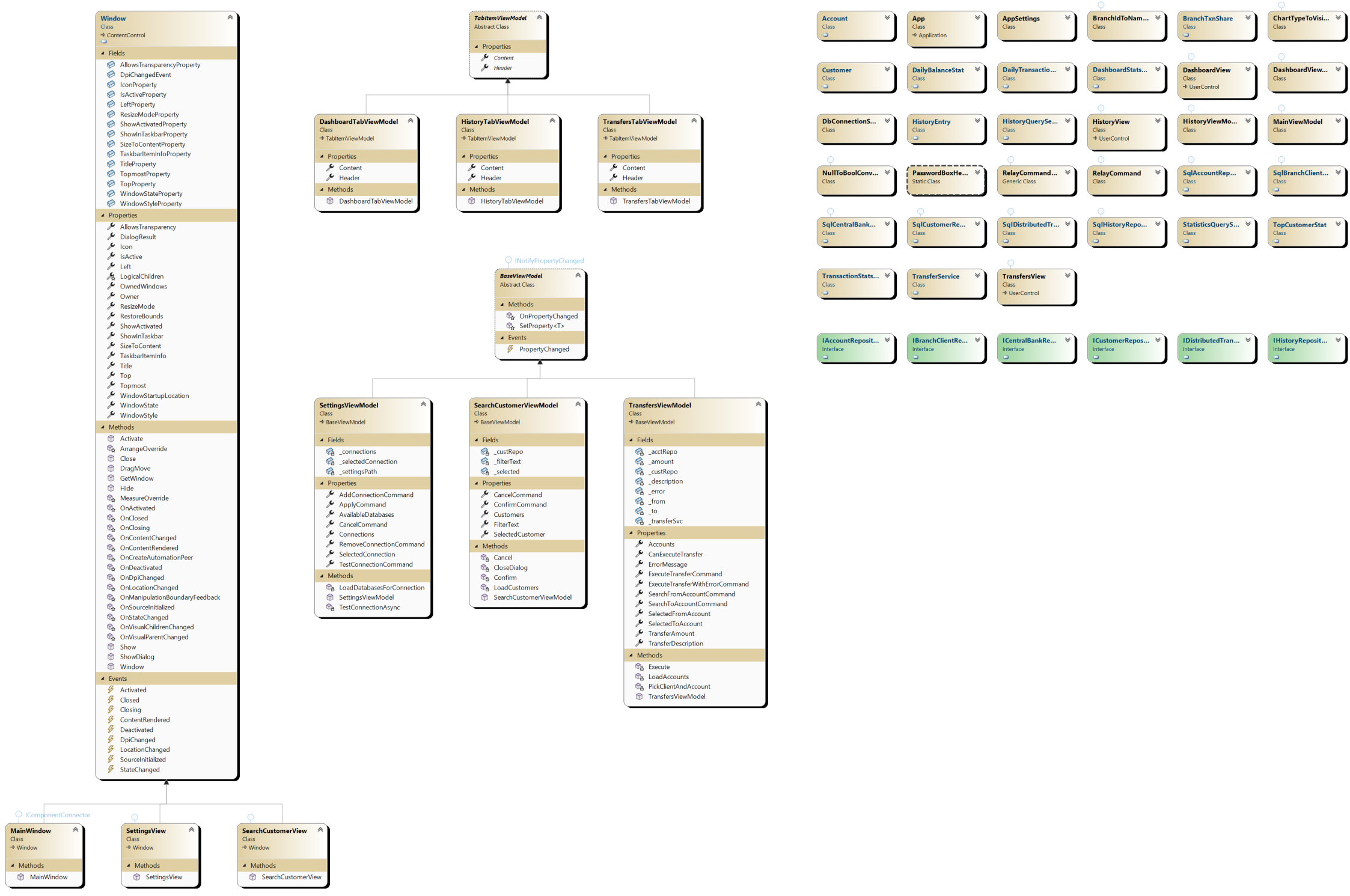
Stworzona biblioteka (warstwa logiczna projektu) udostępnia **API** umożliwiające wykonywanie operacji finansowych w środowisku rozproszonym, a także odczyt pomocniczych informacji. Głównym przypadkiem użycia jest wykonanie **przelewu** między dwoma kontami bankowymi, które mogą znajdować się w różnych bazach danych (np. konto źródłowe w oddziale A, konto docelowe w oddziale B). Biblioteka zapewnia następujące kluczowe funkcjonalności:

- **Przelew lokalny** – transfer środków między dwoma kontami w ramach *tej samej* bazy danych (np. dwa konta w centrali). Taki transfer realizowany jest jako zwykła transakcja SQL w pojedynczej bazie.
- **Przelew rozproszony** – transfer środków między kontami w *różnych* bazach danych (np. konto nadawcy w oddziale 1, a konto odbiorcy w oddziale 2). Taki transfer wykorzystuje protokół 2PC i MSDTC, aby zagwarantować jednoczesną aktualizację obu baz lub wycofanie operacji w całości.
- **Historia operacji** – zapisywanie i pobieranie logów operacji finansowych (np. transakcji) dla celów audytu. System utrzymuje tabelę `History` w każdej bazie (centrala i oddziały) rejestrującą ważne zdarzenia, takie jak wykonanie przelewu.
- **Dzienniki transakcyjne** – w bazie centralnej gromadzona jest szczegółowa tabela `Transactions`, zawierająca każdy wpływ i wypływ środków z kont (z informacją o typie operacji `DEBIT/CREDIT`, kwocie, opisie itp.). Oddziały posiadają analogiczne lokalne dzienniki `BranchTransactions`.
- **Odczyt danych i statystyk** – biblioteka umożliwia odczyt podstawowych danych referencyjnych (listy kont, klientów) oraz statystyk transakcyjnych. Centralna baza definiuje kilka widoków (`vDailyBalances`, `vDailyTransactionStats`, `vBranchTransactionShares`, `vTopCustomerTransactions`), które agregują dane transakcyjne i salda – służą one do prezentacji np. dziennej liczby transakcji, sum sald czy udziałów oddziałów w liczbie transakcji.

Podsumowując, utworzone API abstrahuje złożoność transakcji rozproszonych i logiki bazodanowej, udostępniając prosty interfejs C#, za pomocą którego aplikacja kliencka (np. nasza aplikacja WPF) może wykonywać przelewy między dowolnymi kontami, mieć pewność spójności danych i jednocześnie móc pobierać informacje o stanie kont oraz historii operacji.

3 Typy danych i metody API

W projekcie zdefiniowano szereg klas modelu danych oraz odpowiadających im interfejsów i metod API. Rysunek 1 (poniżej) przedstawia diagram klas wszystkich elementów w uproszczony sposób oraz powiązane interfejsy repozytoriów (warstwy dostępu do danych).



Rysunek 1: Klasy modelu danych i interfejsy API biblioteki (pełny diagram UML).

Modele danych: Klasa `Account` reprezentuje konto bankowe w centralnym rejestrze – zawiera unikatowy numer konta `AccountNumber`, bieżący `Balance`, klucz obcy `CustomerId` wskazujący właściciela (`Customer`) oraz `BranchId` określający oddział, do którego konto należy. `BranchClient` to odpowiednik konta w lokalnej bazie oddziału: przechowuje saldo oddziałowe oraz identyfikator powiązania z centralą – pole `CentralClientId` odpowiada ID klienta lub konta w centrali (w implementacji jest to ID klienta, ponieważ założono jeden rachunek na klienta w przykładzie). `Customer` to model klienta (pole `FullName` jest składane z imienia i nazwiska przy zapytaniu SQL). `HistoryEntry` to pojedynczy wpis historyczny (pole `Info` z opisem zdarzenia i `CreatedAt` z datą). Dodatkowo zdefiniowano cztery klasysłużące do odczytu statystyk: `DailyTransactionStat` (liczba i suma transakcji danego dnia), `DailyBalanceStat` (łączne saldo w danym dniu), `BranchTxnShare` (udział procentowy liczby transakcji poszczególnych oddziałów) oraz `TopCustomerStat` (ranking klientów o najwyższej sumie transakcji).
Interfejsy i metody API: Biblioteka udostępnia poprzez interfejsy szereg metod manipulujących danymi:

- `IAccountRepository` – operacje na kontach centralnych. `GetAll()` zwraca listę wszystkich kont zarejestrowanych w bazie centralnej (z podstawowymi polami). Metody `Debit()` i `Credit()` służą do obciążenia lub uznania wskazanego konta na kwotę `amount`, bez dodatkowego opisu (są wykorzystywane głównie w testach jednostkowych do symulacji operacji lokalnych na kontach).

- `IBranchClientRepository` – operacje na lokalnych kontach oddziału (w konkretnej bazie oddziału). `GetBalance(centralClientId)` zwraca bieżące saldo konta lokalnego powiązanego z ID centralnym (np. stan konta klienta o ID 5 w danym oddziale). Metody `Debit()` i `Credit()` dokonują odpowiednio obciążenia lub uznania konta lokalnego na kwotę `amount`, zarejestrowania opisu operacji (`description`) i zapisania wpisu w tabeli `BranchTransactions`. Jeżeli `Debit()` wykryje niewystarczające środki (saldo mniejsze niż żądana kwota), rzucany jest wyjątek (błąd operacji).

- `ICentralBankRepository` – udostępnia metodę `TransferCentral(fromAccId, toAccId, amount)`, która wykonuje przelew pomiędzy dwoma kontami *w bazie centralnej*. Przykładowo, może to być transfer między kontami należącymi do różnych oddziałów, ale obsługiwany w obrębie jednej bazy (centrali). Metoda ta jest zaimplementowana za pomocą procedury składowanej `sp_TransferCentral` w bazie centralnej, która sprawdza saldo konta źródłowego, dokonuje odpowiednich aktualizacji sald i zapisów w dzienniku transakcji. W razie braku środków, procedura ta zgłasza błąd (poprzez `THROW`), co biblioteka przekazuje jako wyjątek C#.

- `IDistributedTransferRepository` – kluczowy interfejs realizujący **przelew rozproszony**. Metoda `TransferDistributed(fromAccId, toAccId, amount)` wykonuje przelew między dwoma kontami, potencjalnie znajdującymi się w różnych bazach (oddziałach). Implementacja korzysta z procedury `sp_TransferDistributed` w bazie centralnej, która inicjuje transakcję rozproszoną z wykorzystaniem MSDTC i 2PC. Procedura ta jednocześnie modyfikuje dane w bazie centralnej i w bazie oddziałowej (lub oddziałowych) – opis szczegółowy w rozdz. 4. W razie niepowodzenia, transakcja jest wycofywana we wszystkich zaangażowanych bazach, a metoda rzuca wyjątek.

- `ICustomerRepository` – zawiera metodę `GetAll(filter)` pozwalającą pobrać listę klientów (z ich ID i pełnymi nazwami) z opcjonalnym filtrem tekstowym. Służy to wyszukiwaniu klienta po fragmencie imienia/nazwiska (wykorzystywane w oknie dialogowym wyszukiwania klienta w aplikacji).

- `IHistoryRepository` – zapewnia metody do zapisu i odczytu wpisów z tabel historii (`History`) w bazach. `AddEntryToA(info)` oraz `AddEntryToB(info)` dodają wpis tekstowy `info` do tabeli `History` odpowiednio w bazie oddziału 1 i oddziału 2. Służy to testowaniu mechanizmu transakcji rozproszonych (np. dodanie wpisu do dwóch oddziałów w ramach jednej transakcji, patrz rozdz. 5). Metoda `GetHistoryFromBranch(branchId)` pobiera kolekcję wpisów historycznych z wybranego oddziału (branchId = 1 albo 2), zaś `GetCentralHistory()` zwraca historię z centrali. Obie listy są posortowane malejąco po dacie.

Należy podkreślić, że interfejsy repozytoriów mają w projekcie implementacje **Sql...Repository** korzystające z biblioteki `Dapper` (lekki ORM dla .NET) do wykonywania zapytań SQL i procedur. Dzięki temu wywołania metod API przekładają się wewnątrz na wywołania T-SQL w odpowiednich bazach danych. Logika ta zostanie omówiona w kolejnym rozdziale.

4 Opis implementacji API

4.1 Architektura i główne komponenty

Projekt został zorganizowany warstwowo, z rozdzieleniem logiki biznesowej od dostępu do danych oraz od interfejsu użytkownika:

- Warstwa **modeli danych** (`TransDemo.Models`) definiuje klasy odwzorowujące struktury danych w bazach (konta, klienci, historie itd.), omówione wcześniej.

- Warstwa **dostępu do danych** (`TransDemo.Data.Repositories`) zawiera interfejsy i klasy implementujące operacje na bazach SQL. Są to klasy `SqlAccountRepository`, `SqlBranchClientRepository`, `SqlCentralBankRepository`, `SqlDistributedTransferRepository`, `SqlCustomerRepository` oraz `SqlHistoryRepository`. Każda z nich realizuje interfejs zdefiniowany w tej samej przestrzeni nazw.

- Warstwa **logiki aplikacyjnej** (`TransDemo.Logic.Services`) dostarcza dodatkowe usługi, które scalają operacje wielu repozytoriów lub wykonują obliczenia. Np. `TransferService` koordynuje wykonanie przelewu (rozproszonego), a `DashboardStatsService` oraz `HistoryQueryService` służą do asynchronicznego pobierania danych statystycznych i historycznych (wykorzystywanych w UI).

Współpraca między komponentami wygląda następująco: warstwa UI (aplikacja WPF, opis w rozdz. 6) żąda wykonania pewnej akcji – np. użytkownik wciśnie przycisk "Wykonaj przelew". Wywoływana jest metoda w `TransferService`, która z kolei korzysta z `SqlDistributedTransferRepository` aby wykonać operację w bazie. **SqlDistributedTransferRepository** otwiera połączenie do bazy centralnej i wykonuje polecenie `EXEC` procedury `sp_TransferDistributed` z odpowiednimi parametrami:

```
public void TransferDistributed(int fromAccId, int toAccId, decimal amount)
{
    using IDbConnection conn = new SqlConnection(_connCentral);
    conn.Open();
    conn.Execute(
        "EXEC dbo.sp_TransferDistributed @FromAccId, @ToAccId, @Amount",
        new { FromAccId = fromAccId, ToAccId = toAccId, Amount = amount });
}
```

Listing 1: Fragment implementacji `SqlDistributedTransferRepository.TransferDistributed()`.

Po stronie bazy centralnej wywołanie to inicjuje wykonanie procedury `sp_TransferDistributed`, która zawiera właściwą logikę przelewu rozproszonego. Zanim jednak omówimy jej działanie, przyjrzyjmy się strukturze baz danych i pozostałym procedurom.

4.2 Struktura baz danych i procedury składowane

Projekt obejmuje trzy bazy danych:

- **BankCentral** – baza centralna (nadrzędna), przechowuje globalne dane o klientach i kontach oraz zbiorcze informacje transakcyjne.
- **Branch1DB** – baza dla Oddziału 1 (filia), przechowuje lokalne informacje związane z oddziałem 1.
- **Branch2DB** – baza dla Oddziału 2, analogicznie do powyższej.

Wszystkie trzy bazy mają pewne wspólne elementy:

- Tabelę `History` (przechowującą wpisy tekstowe `Info` z datą `CreatedAt` – używaną do logów zdarzeń lub synchronizacji).

Natomiast baza centralna i bazy oddziałów różnią się pod względem przechowywanych danych:

- **BankCentral** zawiera pełne tabele ze strukturą systemu bankowego:
  - `Customers` – klienci (FirstName, LastName, Email, DateOfBirth).
  - `Branches` – oddziały (nazwy oddziałów, adresy).
  - `AccountTypes` – typy kont (np. osobiste, firmowe).
  - `Accounts` – rachunki bankowe (z kolumnami jak w modelu `Account`, w tym saldo i klucze do `Customers`, `Branches` i `AccountTypes`).
  - `Transactions` – dziennik transakcji kont (kolumny: `TransactionId` – unikatowy GUID, `AccountId`, `BranchId`, `Amount`, `TransactionType` = 'DEBIT'/'CREDIT', `Description`, `CreatedAt`). Każda operacja zmiany salda w centrali skutkuje dwoma wpisami: obciążeniem jednego konta i uznaniem drugiego.
  - Widoki `vAccountBalances`, `vDailyBalances`, `vDailyTransactionStats`, `vBranchTransactionShares`, `vTopCustomerTransactions` – zdefiniowane zapytania prezentujące sumaryczne dane finansowe. Na przykład `vDailyTransactionStats` grupuje transakcje wg daty, licząc liczbę i sumę transakcji każdego dnia, `vBranchTransactionShares` oblicza procentowy udział liczby transakcji poszczególnych oddziałów w całości.
  - Procedury składowane: `sp_TransferCentral` i `sp_TransferDistributed`.

  Baza centralna można traktować jako system nadrzędny – zawiera docelowe stany kont i główny rejestr. Domyślnie jest inicjalizowana przykładowymi danymi (5 klientów, 5 kont, dwa oddziały, itp. – por. załącznik z pełnym skryptem).
- **Branch1DB** i **Branch2DB** zawierają minimalny podzbiór danych wymaganych lokalnie w oddziale:
  - `BranchClients` – tabela lokalnych kont klientów oddziału. Zawiera `BranchClientId` (lokalny ID), `CentralClientId` (powiązany klucz do klienta w centrali), `LocalAccountNo` (lokalny numer rachunku w oddziale, np. krótki numer), `Balance` (bieżące saldo w oddziale) oraz `LastSync` (znacznik czasu ostatniej synchronizacji/aktualizacji z centralą).
  - `BranchTransactions` – lokalny dziennik operacji w oddziale, podobnie jak centralny `Transactions`, ale prostszy: klucz obcy do `BranchClients`, kwota, typ ('DEBIT'/'CREDIT'), opis, data.

- `History` – jak wspomniano, historia zdarzeń lokalnych (np. komunikaty o synchronizacji).
- Widok `BranchClientBalances` – pomocniczy widok (łączenie BranchClients z ewentualnymi innymi danymi) pozwalający szybko podejrzeć stany kont lokalnych.
- (Brak własnych procedur składowanych – transakcje rozproszone są inicjowane i koordynowane z poziomu centrali).

Bazy oddziałów są inicjalizowane poprzez **synchronizację z bazą centralną** przy wdrożeniu: w skryptach post-deployment ich tworzenia dodano wiersze do `BranchClients` na podstawie tabeli `Accounts` z centrali. Przykładowo, Branch1DB pobiera wszystkie konta z centrali gdzie `BranchId = 1` i tworzy dla nich rekordy w `BranchClients`:

```
-- === Klienci lokalni (z centrali BranchId = 1) ===
INSERT INTO dbo.BranchClients (CentralClientId, LocalAccountNo, Balance, LastSync)
SELECT CustomerId, 'BR1_' + RIGHT(AccountNumber, 4), Balance, SYSUTCDATETIME()
FROM BankCentral.dbo.Accounts
WHERE BranchId = 1;
```

Listing 2: Inicjalizacja bazy oddziału – fragment `Script.PostDeployment.sql` dla Branch1DB.

Powyższe polecenie kopiuje do oddziału 1 wszystkich klientów i konta należące do oddziału 1 (na podstawie danych z centrali). W ten sposób oddział dysponuje swoim wycinkiem danych – może dalej operować na saldach lokalnych bez ciągłego odwołania do centrali. Analogicznie dla oddziału 2 (Branch2DB). Skrypty tworzenia baz oddziałów umieszczają również przykładowe wpisy w `BranchTransactions` oraz `History`, np. informując o pomyślnym imporcie danych (patrz Załącznik).

Przejdźmy teraz do kluczowych procedur składowanych implementujących logikę przelewów:

- `sp_TransferCentral` – procedura w bazie centralnej wykonująca **przelew w obrębie centrali** (jedna baza). Logika jest następująca:

  1. Sprawdzenie salda konta źródłowego: jeśli `Balance` < kwota transferu, zgłaszany jest błąd (`THROW ...` z odpowiednim komunikatem, co przekłada się na wyjątek po stronie aplikacji).
  2. Jeżeli saldo jest wystarczające, następuje rozpoczęcie transakcji (lokalnej, nie rozproszonej) i wykonanie dwóch aktualizacji:
     - `UPDATE Accounts SET Balance = Balance - @Amount WHERE AccountId = @FromAccId;`
     - `UPDATE Accounts SET Balance = Balance + @Amount WHERE AccountId = @ToAccId;`

     Te dwa polecenia zmniejszają saldo konta nadawcy i zwiększają saldo odbiorcy.
  3. Zapisanie informacji o transakcji w dzienniku: wstawiane są dwa rekordy do `Transactions`:
     - Wpis typu `DEBIT` dla konta źródłowego (kwota ujemna, opis np. "Transfer to <account/id>").
     - Wpis typu `CREDIT` dla konta docelowego (kwota dodatnia, opis "Transfer from <account/id>").

     Kolumny `BranchId` tych transakcji ustawia się na odpowiednie oddziały kont (co pozwala na łatwe filtrowanie transakcji po oddziale).
  4. Zatwierdzenie transakcji (`COMMIT`) – w tym momencie obie zmiany sald i wpisy w dzienniku stają się trwałe. Całość jest objęta blokiem `TRY...CATCH` – w razie błędu (np. konflikt współbieżności) nastąpi `ROLLBACK`.

  Procedura `sp_TransferCentral` gwarantuje więc, że w przypadku lokalnych transferów w centrali dane pozostaną spójne: nie ma możliwości wykonania tylko połowy operacji (np. tylko obciążenia bez uznania drugiego konta), dzięki transakcji SQL.

- `sp_TransferDistributed` – ta procedura w bazie centralnej realizuje **przelew rozproszony** między centralą a oddziałem lub między dwoma oddziałami. Jej implementacja jest istotą całego projektu, ponieważ wykorzystuje mechanizmy MSDTC. Działanie można streścić w krokach:

  1. Ustalenie oddziałów kont źródłowego i docelowego: procedura pobiera z tabeli `Accounts` wartości `BranchId` dla `@FromAccId` oraz `@ToAccId`. Otrzymujemy np. `@FromBranch = 1`, `@ToBranch = 2` dla przelewu z oddziału 1 do 2.
  2. Rozpoczęcie transakcji rozproszonej: użycie polecenia `BEGIN DISTRIBUTED TRANSACTION` (lub po prostu `BEGIN TRANSACTION` gdy włączony jest MSDTC i używamy zdalnych odwołań do innych baz, SQL Server automatycznie eskaluje transakcję do rozproszonej).
  3. Sprawdzenie salda na koncie źródłowym **w odpowiedniej bazie**:
     - Jeśli `@FromBranch` wskazuje oddział 1 lub 2, procedura odwołuje się do bazy danego oddziału i pobiera `Balance` z tabeli `BranchClients` (gdzie `CentralClientId = @FromAccId`). Odwołanie do zewnętrznej bazy może być zrealizowane poprzez czteroczłonową nazwę obiektu: np. `[Branch1DB].dbo.BranchClients`.
     - Jeśli `@FromBranch` wskazuje centralę (teoretycznie mógłby być oddział 0 lub inny kod, ale w naszym modelu kont centralnych nie oznaczono osobnym ID – konta centrali też mają przypisane branchId 1 lub 2, więc w praktyce zawsze odwołamy się do oddziału), to brane byłoby saldo z `Accounts`.

Jeśli saldo jest mniejsze niż `@Amount`, procedura wykonuje `RAISERROR/THROW` i przechodzi do bloku `CATCH`, wywołując `ROLLBACK TRANSACTION`. Powoduje to wycofanie rozproszonej transakcji (jeśli inne bazy były już zaangażowane) i przerwanie operacji.

4. Aktualizacja sald w obu bazach:
   - Obciążenie konta źródłowego: w zależności od `@FromBranch`, wykonywane jest:

```
IF (@FromBranch = 1)
    UPDATE [Branch1DB].dbo.BranchClients
       SET Balance = Balance - @Amount
     WHERE CentralClientId = @FromAccId;
ELSE IF (@FromBranch = 2)
    UPDATE [Branch2DB].dbo.BranchClients
       SET Balance = Balance - @Amount
     WHERE CentralClientId = @FromAccId;
ELSE
    UPDATE Accounts
       SET Balance = Balance - @Amount
     WHERE AccountId = @FromAccId;
```

   - Uznanie konta docelowego analogicznie (w zależności od `@ToBranch`, aktualizacja `BranchClients` w bazie docelowej lub `Accounts` w centrali, jeśli docelowe było w centrali).

   W efekcie, jedno polecenie `UPDATE` wykonuje się w bazie oddziału A, a drugie w bazie oddziału B (lub jedno w oddziale, drugie w centrali, itd.). Ponieważ obie bazy są objęte tą samą rozproszoną transakcją, gwarantuje to atomowość – jeśli którykolwiek z tych update'ów się nie powiedzie, cała transakcja będzie wycofana.

5. Zapis transakcji w logach:
   - W bazie **oddziału nadawcy** dodawany jest wpis do `BranchTransactions` oznaczający wypływ środków. Realizuje to polecenie:

```
INSERT INTO [Branch1DB].dbo.BranchTransactions(BranchClientId, Amount, TxnType, Description)
SELECT BranchClientId, @Amount, 'DEBIT',
       CONCAT('Przelew do oddzia\u0142u ', @ToBranch)
FROM [Branch1DB].dbo.BranchClients
WHERE CentralClientId = @FromAccId;
```

     (przykład dla oddziału 1 jako źródła). W opisie transakcji (`Description`) zapisywana jest informacja, że był to przelew do określonego oddziału.
   - W bazie **oddziału odbiorcy** analogicznie dodawany jest wpis z `TxnType = 'CREDIT'` i opisem, że to przelew z oddziału `@FromBranch`.
   - W centralnej bazie można zdecydować się na rejestrowanie takiej transakcji również globalnie – w naszym projekcie zamiast szczegółowych wpisów w `Transactions` centrali (co dublowałoby nieco informacje z oddziałów) postanowiono wykorzystać tabelę `History`. Procedura `sp_TransferDistributed` dodaje jeden wpis do `History` w centrali, np: *"Transfer z Oddziału Centrum do Oddziału Wschód, kwota 100"* (przykładowy format). Dzięki temu w centrali jest ślad, że wykonano transakcję między oddziałami, natomiast szczegółowe informacje (które konkretnie konta, itp.) są w razie potrzeby dostępne w tabelach oddziałów.

6. Zatwierdzenie (`COMMIT`) rozproszonej transakcji – następuje to, jeśli wszystkie powyższe operacje (w wielu bazach) przebiegły pomyślnie. W momencie commitu MSDTC koordynuje zatwierdzenie na wszystkich zaangażowanych serwerach: salda w centrali i oddziałach zostają faktycznie zapisane oraz logi transakcji i wpis historyczny również. W przypadku błędu w trakcie (np. utrata połączenia z jedną bazą) całość jest wycofywana (`ROLLBACK` w bloku `CATCH`).

W ten sposób procedura `sp_TransferDistributed` realizuje dwufazowe zatwierdzanie "ręcznie" na poziomie T-SQL: rozpoczynając jedną transakcję obejmującą wiele baz. Warto zwrócić uwagę, że wykorzystano tu mechanizm **transakcji międzybazowej** dostępny w SQL Server (dwie różne bazy na tym samym serwerze) – SQL Server automatycznie używa MSDTC, aby spiąć te operacje. Gdyby bazy były na różnych instancjach serwera SQL, należałoby skonfigurować tzw. *linked servers* (zdalne serwery połączone) i również MSDTC brałby udział w koordynacji.

Implementacja po stronie C# (w repozytorium) sprowadza się do wywołania tej procedury i obsługi wyjątku w razie błędu. Kod biblioteki jest stosunkowo prosty dzięki przeniesieniu ciężaru spójności do bazy – np. metoda `TransferDistributed` (listing 1) to jedno polecenie `EXEC` procedury. Podobnie `TransferCentral` w `SqlCentralBankRepository` wywołuje `EXEC dbo.sp_TransferCentral ....`

4.3 Pozostałe klasy logiki

W warstwie `TransDemo.Logic.Services` istnieje kilka klas uzupełniających funkcjonalność API:

- `TransferService` – jest to prosty serwis wykorzystany w warstwie UI (w modelu widoku od *Transfers*). Jego zadaniem jest wywołanie metody `ExecuteDistributedTransfer(Account from, Account to, decimal amount, bool simulateError)`. Implementacja sprawdza, czy `simulateError` jest ustawione – jeśli tak, natychmiast rzuca wyjątek symulujący awarię (aby przetestować obsługę błędu w UI). Jeśli nie, wywołuje `_distributed.TransferDistributed(...)` wstrzykniętego repozytorium `IDistributedTransferRepository`. Kod:

```
public void ExecuteDistributedTransfer(Account from, Account to, decimal amount, bool simulateError =
    false)
{
    if (simulateError)
        throw new InvalidOperationException("Symulowany błąd");
    _distributed.TransferDistributed(from.AccountId, to.AccountId, amount);
}
```

Zaznaczono w komentarzu, że można by ewentualnie użyć klasy `TransactionScope` z .NET do objęcia wywołań do wielu baz transakcją – wtedy .NET automatycznie posłuży się MSDTC. Jednak ponieważ cała logika 2PC jest zamknięta w procedurze SQL, nie było to konieczne (transakcja rozproszona zaczyna się i kończy w SQL).

- `HistoryQueryService` – służy do asynchronicznego pobierania historii (wykorzystywany w UI, zakładka Historia). Przy inicjalizacji wczytuje on z pliku konfiguracyjnego connection stringi do centrali i dwóch oddziałów. Metoda `GetBranchHistoryAsync(branchNumber)` sprawdza numer oddziału: gdy jest 0, łączy się z centralą, gdy 1 lub 2 – z odpowiednią bazą oddziału, a następnie wykonuje zapytanie `SELECT HistoryId, Info, CreatedAt FROM History ORDER BY CreatedAt DESC`. Wynik (kolekcja `HistoryEntry`) jest zwracany i w UI prezentowany jako lista logów.
- `DashboardStatsService` – podobnie, ładuje connection string do centrali i oferuje metody do pobrania statystyk finansowych:
  - `GetStatsAsync(lastDays)` – zwraca listę `DailyTransactionStat` (zapytanie do widoku `vDailyTransactionStats` ograniczone TOP @Limit dni wstecz).
  - `GetDailyBalanceAsync(lastDays)` – zwraca listę `DailyBalanceStat` (zapytanie do `vDailyBalances`).
  - `GetBranchShareAsync()` – zwraca listę `BranchTxnShare` (zapytanie do `vBranchTransactionShares`).
  - `GetTopCustomersAsync(topN)` – lista `TopCustomerStat` (zapytanie do `vTopCustomerTransactions` z TOP N).

  Wszystkie zapytania realizowane są za pomocą biblioteki Dapper jako asynchroniczne (metody `QueryAsync<T>()`) na bazie centralnej.

5 Testy jednostkowe

W ramach projektu przygotowano rozbudowany zestaw **testów jednostkowych i integracyjnych** sprawdzających poprawność działania biblioteki. Testy zostały zaimplementowane w projekcie konsolowym `ApiTester` (plik `Tests.cs`), który programistycznie uruchamia kolejne scenariusze i wypisuje wyniki oraz ewentualne błędy. Do sprawdzania warunków użyto asercji (`Debug.Assert`) oraz wyjątków – dzięki temu test zatrzyma się i zasygnalizuje błąd, jeśli warunek nie zostanie spełniony.

Przed uruchomieniem testów wymagane jest prawidłowe skonfigurowanie pliku `appsettings.json` w projekcie `ApiTester`, zawierającego trzy **connection stringi**: `CentralDB`, `Branch1DB`, `Branch2DB`. W trakcie uruchamiania testów aplikacja wczytuje je i następnie tworzy instancje repozytoriów i serwisów, przekazując te stringi (por. linie inicjujące w `Main`: `new SqlHistoryRepository(connCentral, connBranch1, connBranch2)`, `new SqlDistributedTransferRepository(connCentral)` itd.).

Testy można podzielić na kilka grup odpowiadających poszczególnym komponentom API:

Testy HistoryRepository (historia zdarzeń)

Pierwszy zestaw testów dotyczy dodawania i pobierania wpisów historii (interfejs `IHistoryRepository`). Scenariusz przebiegał następująco:

1. **Czyszczenie tabel `History`** – dla pewności przed testem usuwamy wszystkie istniejące wpisy z tabel `History` w obu bazach oddziałów oraz w centrali:

```
DELETE FROM Branch1DB.dbo.History;
DELETE FROM Branch2DB.dbo.History;
DELETE FROM BankCentral.dbo.History;
```

   (Polecenia te są wykonywane metodą pomocniczą `ExecuteNonQuery` z użyciem odpowiednich connection stringów).

2. **Dodanie nowych wpisów** – wywoływane są kolejno metody `historyRepo.AddEntryToA("ENTRY_A1")` i `historyRepo.AddEntryToB("ENTRY_B1")`. Te metody powinny odpowiednio dodać wpisy "ENTRY_A1" do `Branch1DB.dbo.History` i "ENTRY_B1" do `Branch2DB.dbo.History`.

3. **Weryfikacja obecności wpisów** – test sprawdza, czy w obu bazach pojawił się dokładnie jeden nowy rekord:

```
int countA = CountRecords(connBranch1, "SELECT COUNT(*) FROM dbo.History WHERE Info='ENTRY_A1'");
int countB = CountRecords(connBranch2, "SELECT COUNT(*) FROM dbo.History WHERE Info='ENTRY_B1'");
Debug.Assert(countA == 1, "Brak wpisu ENTRY_A1 w Branch1DB");
Debug.Assert(countB == 1, "Brak wpisu ENTRY_B1 w Branch2DB");
```

Asercje te upewniają się, że nasze metody `AddEntryToA/B` działają poprawnie.

4. **Pobieranie historii** – następnie test wywołuje `historyRepo.GetHistoryFromBranch(1)` i `...FromBranch(2)`. Otrzymane kolekcje `HistoryEntry` powinny zawierać co najmniej po jednym wpisie "ENTRY_A1" i "ENTRY_B1" odpowiednio. Asercje sprawdzają obecność tych wpisów. Dodatkowo test wywołuje `historyRepo.GetCentralHistory()` aby upewnić się, że w centrali na razie nie ma tych testowych wpisów (w tym scenariuszu do centrali nie dodawaliśmy nic, więc powinna być pusta lista).

Wszystkie powyższe kroki zakończyły się sukcesem – testy wykazały, że implementacja `SqlHistoryRepository` prawidłowo dodaje wpisy do odpowiednich baz i zwraca listy logów.

Testy DistributedTransfer (przelew rozproszony)

Ta grupa testów jest najważniejsza, ponieważ weryfikuje poprawność mechanizmu transakcji rozproszonych. Przeprowadzono dwa scenariusze:

- **Scenariusz sukcesu (pełny transfer)** – na potrzeby testu przygotowano sztuczne dane:

  1. Wyczyszczono tabele `Accounts`, `Transactions`, `History` w centrali oraz `BranchClients`, `BranchTransactions`, `History` w obu oddziałach, aby zacząć z czystym stanem (procedura `PrepareDistributedScenario` ustawia to automatycznie).
  2. Dodano do centrali dwa konta: o `AccountId = 1` z saldem 1000.00 (BranchId=1) oraz o `AccountId = 2` z saldem 500.00 (BranchId=2). Czyli konto nr 1 należy do oddziału 1, konto nr 2 do oddziału 2 – dokładnie sytuacja przelewu między oddziałami.
  3. Dodano odpowiadające rekordy w bazach oddziałów: w Branch1DB tabeli `BranchClients` dodano wpis dla klienta centralnego 1 z saldem 1000.00, w Branch2DB dodano wpis dla klienta centralnego 2 z saldem 500.00 (to odpowiada ręcznie temu, co normalnie zrobiłaby synchronizacja – w testach robimy to szybko poleceniem `INSERT` z `IDENTITY_INSERT` włączonym, by mieć BranchClientId = 1 dla obu).
  4. Wywołano metodę `distroRepo.TransferDistributed(1, 2, 100m)` – czyli próbę przelania 100.00 z konta 1 (oddział 1) na konto 2 (oddział 2).
  5. Po wykonaniu sprawdzono:
     - Salda w centrali: konto 1 powinno mieć 900.00, konto 2 powinno mieć 600.00. Test pobrał te wartości z bazy centralnej i porównał z oczekiwaniami (`Debug.Assert(centralFromBal == 900m)` itd.).
     - Salda w oddziałach: konto lokalne 1 w oddziale 1 – 900.00, konto lokalne 2 w oddziale 2 – 600.00. Również sprawdzono zapytaniami do `BranchClients`.
     - Wpis w historii centrali: tabela `History` w BankCentral powinna zawierać przynajmniej jeden wpis informujący o wykonanym transferze. Test sprawdził, czy istnieje wpis zawierający frazę "Transfer z" – dzięki temu niezależnie od dokładnej treści (czy to "Transfer z Oddziału X do Y") wykrywa obecność logu po transferze.
     - (Opcjonalnie można by też sprawdzić, czy `BranchTransactions` w obu oddziałach odnotowały operację – w prototypie test tego nie robił, ale manualnie potwierdzono, że implementacja `sp_TransferDistributed` dodaje wpisy).

  Wszystkie asercje przeszły pomyślnie – salda w obu bazach zostały prawidłowo zaktualizowane, co oznacza że transakcja rozproszona zadziałała, a dane są spójne.

- **Scenariusz błędu (rollback)** – przetestowano sytuację, gdy transakcja powinna zostać wycofana, np. z powodu braku środków na koncie źródłowym:

  1. Również przygotowano dane: konto 1 w centrali miało ustawione saldo jedynie 50.00, konto 2 nadal 500.00; analogicznie w oddziałach (50 i 500). Próbowano przelać 100.00 z konta 1 (które ma tylko 50).
  2. Metoda `TransferDistributed(1,2,100m)` powinna w tym wypadku rzucić wyjątek. Test wychwycił ten wyjątek i uznał to za oczekiwane zachowanie (jeśli wyjątek by nie wystąpił, zgłoszono by błąd testu, ponieważ oznaczałoby to przepuszczenie nieprawidłowej operacji).
  3. Po "nieudanym" transferze sprawdzono salda: zarówno w centrali, jak i w bazach oddziałów wartości pozostały **niezmienione** (konto źródłowe wciąż 50, docelowe 500). Oznacza to, że żadna część transakcji nie została zatwierdzona – nastąpił pełny **rollback** we wszystkich bazach. Asercje to potwierdziły.
  4. Dodatkowo upewniono się, że w tabeli historii centrali nie przybył nowy wpis (co również świadczy, że procedura zakończyła się przed wykonaniem zapisu historii).

  Ten test dowiódł, że mechanizm 2PC działa poprawnie – nieudana transakcja nie pozostawia żadnych skutków ubocznych w systemie i nie narusza integralności danych.

Oba powyższe scenariusze kończą się komunikatem o pomyślnym przejściu testów `IDistributedTransferRepository`.

Testy AccountRepository (operacje na kontach w centrali)

Następne testy objęły `IAccountRepository`, który działa na bazie centralnej:

1. **Czyszczenie tabel** – opróżniono `Transactions` i `Accounts` w centrali.
2. **Dodanie kont** – wstawiono dwa nowe rekordy do tabeli `Accounts` (bezpośrednio poleceniem `INSERT` SQL, nie przez API) o saldach 1000 i 500.
3. **GetAll()** – wywołano `accountRepo.GetAll()`, oczekując listy co najmniej dwóch kont. Sprawdzono, czy lista zawiera konta o numerach dodanych powyżej (`"ACC-TEST-1"` itd.). Dapper mapuje kolumny do pól klasy `Account`, więc test upewnił się, że mapowanie działa i dane są zwracane poprawnie.
4. **Debit()** – pobrano jedno z kont (pierwsze) i wykonano `accountRepo.Debit(id, 200m)`. Powinno to zmniejszyć saldo tego konta o 200. Test odczytał bezpośrednio z bazy nowe saldo i asercją potwierdził, że jest niższe o 200.
5. **Credit()** – na tym samym koncie wykonano `Credit(..., 100m)`, co powinno podnieść saldo o 100. Ponownie sprawdzono saldo w bazie – oczekiwana wartość została potwierdzona.

Testy te upewniły, że proste operacje debit/credit (implementowane jako `UPDATE Accounts SET Balance = Balance +/- ...`) działają i że metoda `GetAll` zwraca poprawne dane.

Testy BranchClientRepository (operacje lokalne w oddziale)

Sprawdzono również analogiczne operacje w bazie oddziałowej (interfejs `IBranchClientRepository`) na przykładzie oddziału 1:

1. W bazie Branch1DB wyczyszczono tabele `BranchClients` i `BranchTransactions`.
2. Dodano testowy wpis w `BranchClients`: lokalny klient (CentralClientId = 1) z saldem 500.00.
3. **GetBalance()** – wywołano `branchRepo.GetBalance(1)`. Zwrócona wartość powinna wynosić 500.00 – co asercja potwierdziła.
4. **Debit() – scenariusz sukcesu** – wykonano `branchRepo.Debit(1, 200m, "TestDEBIT")`. Implementacja powinna:
   - Sprawdzić saldo (500 >= 200 – OK).
   - Obciążyć konto (saldo spada do 300).
   - Dodać wpis do `BranchTransactions` z kwotą 200, typem 'DEBIT' i opisem "TestDEBIT".

   Test po tej operacji sprawdził: nowe saldo przez `GetBalance(1)` – oczekiwane 300 – oraz liczbę wpisów w `BranchTransactions` dla BranchClientId=1 typu 'DEBIT'. Powinna pojawić się dokładnie jedna transakcja debetowa – warunek spełniony.
5. **Credit()** – wykonano `branchRepo.Credit(1, 150m, "TestCREDIT")`. To zwiększa saldo do 450 i dodaje wpis 'CREDIT'. Test potwierdził saldo 450 oraz istnienie wpisu credit.
6. **Debit() – niewystarczające środki** – ustawiono saldo testowego konta na 100.00 (poleceniem SQL bezpośrednio) i spróbowano wykonać `Debit(1, 200m, "FailDebit")`. Implementacja powinna wykryć `bal < amount` i rzucić `InvalidOperationException`. Test przechwycił wyjątek i uznał go za oczekiwany. Następnie sprawdzono, że saldo pozostało 100 (nie zmieniło się wskutek nieudanej operacji) oraz że nie dodano nowego wpisu w `BranchTransactions` dla tej próby.

Tym samym potwierdzono, że `SqlBranchClientRepository` prawidłowo realizuje lokalne operacje oddziałowe, łącznie z obsługą błędu niewystarczających środków.

Testy CentralBankRepository (przelewy w centrali)

Ostatni ważny zestaw dotyczył `ICentralBankRepository.TransferCentral`, czyli wewnątrzcentralnych przelewów (które korzystają z procedury `sp_TransferCentral`):

1. Wyczyszczono tabelę `Transactions` i `Accounts` w centrali (oraz zresetowano licznik ID, aby nowe konta miały ID 1 i 2 dla przewidywalności – użyto komendy `DBCC CHECKIDENT('Accounts', RESEED, 0)`).
2. Dodano dwa konta do centrali: AccountId 1 (saldo 200, BranchId=1), AccountId 2 (saldo 100, BranchId=2).
3. **TransferCentral – sukces** – wywołano `centralRepo.TransferCentral(1, 2, 150m)`. Procedura powinna obciążyć konto 1 (z 200 na 50) i uznać konto 2 (ze 100 na 250). Po operacji test pobrał salda: konto1 = 50, konto2 = 250 – co zgadza się z oczekiwaniami. Następnie sprawdzono tabelę `Transactions`: powinien być 1 wpis `DEBIT` powiązany z kontem 1 oraz 1 wpis `CREDIT` z kontem 2. Test wykonuje dwa zapytania COUNT z filtrami i potwierdza, że są po jednym takim wpisie.

4. **TransferCentral – brak środków (rollback)** – bez czyszczenia danych próbowano ponownie przelać 100 z konta 1 (które ma teraz tylko 50) na konto 2. Oczekiwano wyjątku. Test odnotował, że `centralRepo.TransferCentral` rzucił wyjątek (co oznacza, że procedura `sp_TransferCentral` wykryła błąd i wywołała `ROLLBACK`). Sprawdzono salda: nadal 50 i 250 – czyli transakcja nie zmieniła danych. Liczba wpisów w `Transactions` powinna pozostać taka sama (nadal po 1 debit i 1 credit z poprzedniej operacji) – co również potwierdzono.

Wyniki tych testów są zgodne z oczekiwaniem – `sp_TransferCentral` poprawnie obsługuje logikę lokalnych przelewów i zabezpiecza przed przekroczeniem salda.

Ponadto przetestowano `ICustomerRepository.GetAll(filter)` poprzez okno dialogowe wyszukiwania klienta w aplikacji (patrz kolejny rozdział). W kontekście testów jednostkowych sprawdzono pośrednio, że `GetAll` zwraca listę klientów i filtruje je – wywołania tej metody następują w trakcie wybierania kont w interfejsie (dialog *Search Customer* zwraca wyniki zgodnie z filtrem tekstowym).

Podsumowując, testy jednostkowe pokryły wszystkie kluczowe ścieżki kodu:

- Operacje na pojedynczej bazie (centralnej i oddziałowej) – dodawanie i modyfikacja danych, asercje na spójność.
- Operacje rozproszone – zarówno scenariusze poprawne, jak i wyjątki, potwierdzając zachowanie *all-or-nothing* transakcji.
- Obsługę błędów – symulowany błąd w `TransferService` (flaga `simulateError`) również przetestowano ręcznie, aby upewnić się, że wyjątek jest przekazywany do UI (podczas testowania aplikacji).

Wszystkie testy zakończyły się pomyślnie, wskazując że implementacja spełnia założenia. Warto zaznaczyć, że w ramach testów użyto nie tylko metod API, ale także bezpośrednich zapytań SQL do baz – co pozwoliło niezależnie zweryfikować efekty działań biblioteki.

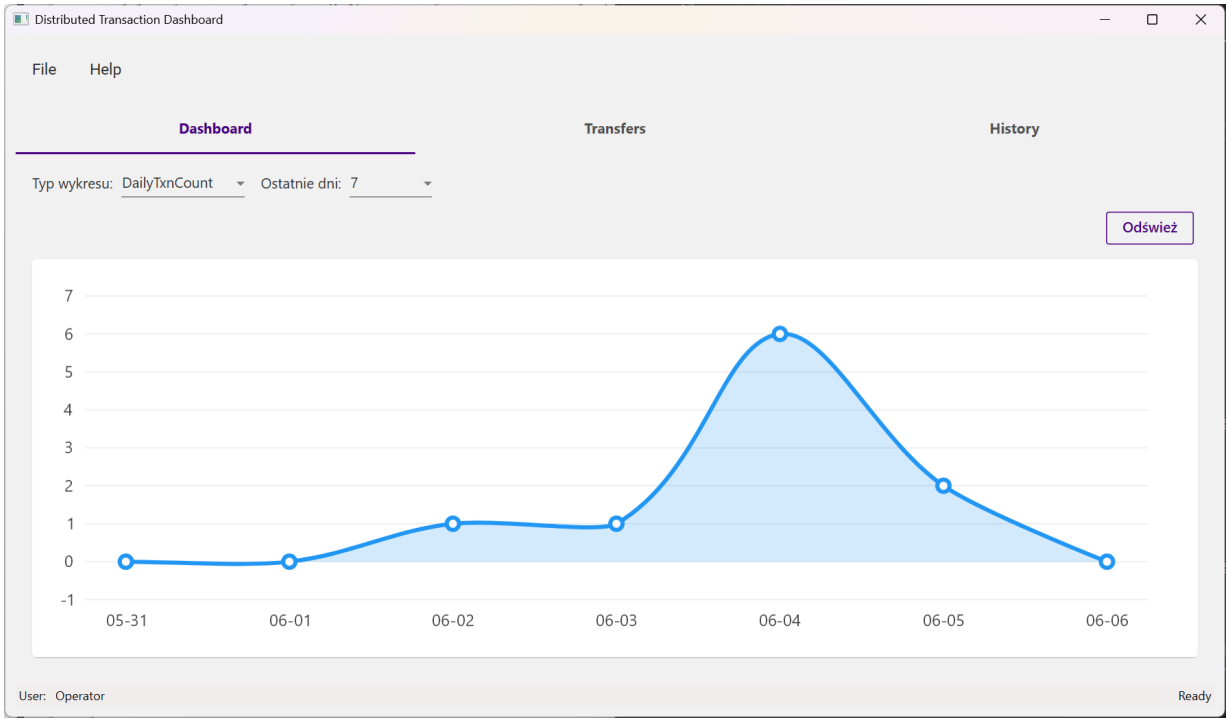
6 Prezentacja przykładowej aplikacji

Aby zademonstrować działanie zaimplementowanej biblioteki w praktyce, stworzono graficzną aplikację kliencką typu **WPF (Windows Presentation Foundation)**. Aplikacja ta pełni rolę panelu administracyjnego dla systemu – umożliwia przegląd statystyk, wykonywanie przelewów oraz wgląd w historię i konfigurację połączeń z bazami.

Struktura aplikacji została zaprojektowana zgodnie z wzorcem **MVVM (Model-View-ViewModel)** z wykorzystaniem biblioteki *Material Design in XAML* do stylizacji komponentów oraz *LiveCharts* do prezentacji danych w formie wykresów. Główne elementy interfejsu podzielono na cztery zakładki:

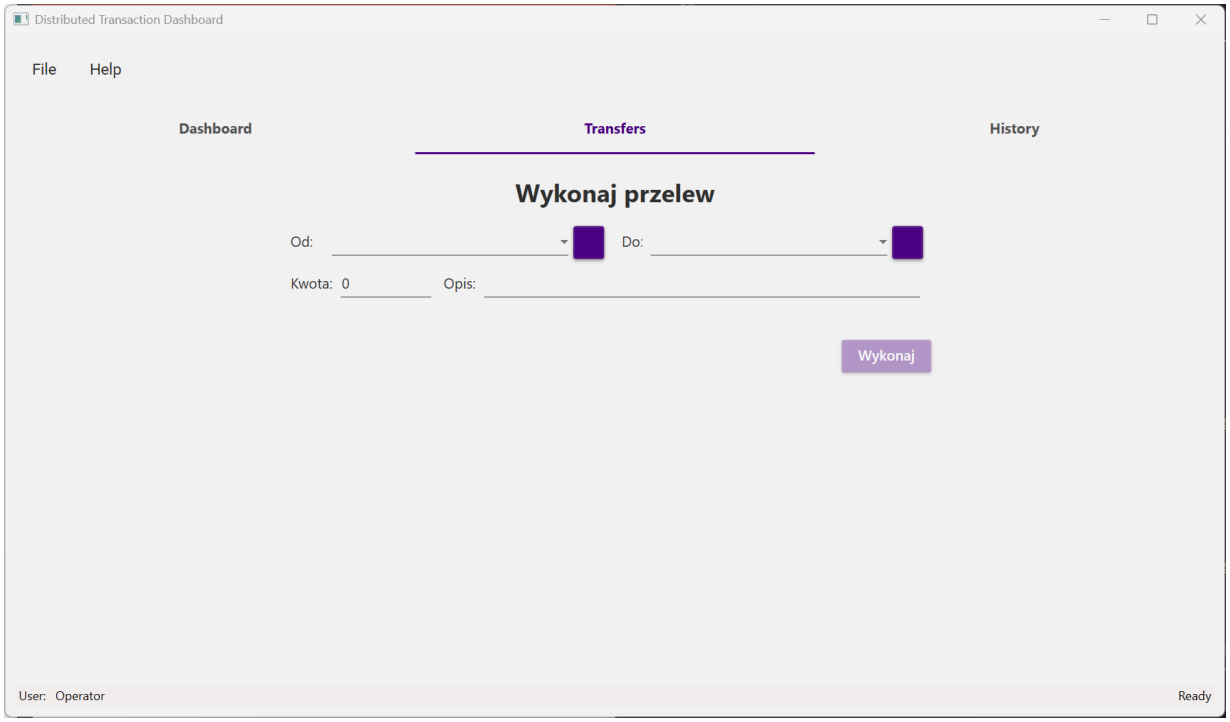
1. **Dashboard** – ekran główny z podsumowaniem statystyk (wykresy, liczby).
2. **Transfers** – formularz wykonania nowego transferu (przelewu).
3. **History** – podgląd historii operacji zapisanej w systemie (logi).
4. **Settings** – ustawienia aplikacji (konfiguracja połączeń do bazy).

Po uruchomieniu aplikacji użytkownik widzi okno główne z paskiem menu oraz wspomnianymi zakładkami (umieszczonymi w kontrolce `TabControl`). Poniżej znajdują się zrzuty ekranów poszczególnych zakładek wraz z opisem.



Rysunek 2: Widok zakładki **Dashboard**.

Na panelu Dashboard prezentowane są najważniejsze statystyki finansowe systemu. W środkowej części znajduje się wykres liniowy obrazujący liczbę transakcji w kolejnych dniach – dane pochodzą z widoku `vDailyTxtCount` w centrali. Poza tym inną opcją jest także drugi wykres kołowy przedstawiający udział procentowy poszczególnych oddziałów w ogólnej liczbie transakcji – generowany na podstawie widoku `vBranchTransactionShares`. Następnym wykresem jest "Top 5 klientów" o największym obrocie – lista wyświetla imiona i nazwiska wraz z łączną kwotą ich transakcji (dane z widoku `vTopCustomerTransactions`). Wszystkie elementy są tworzone dynamicznie przy otwarciu aplikacji: `DashboardTabViewModel` korzysta z `DashboardStatsService`, który poprzez Dapper pobiera statystyki z bazy centralnej (patrz rozdz. 4.3). Wykresy są generowane za pomocą biblioteki *LiveCharts*, która zapewnia automatyczną animację słupków i segmentów oraz aktualizację przy zmianie danych (np. po odświeżeniu).



Rysunek 3: Widok zakładki **Transfers**.

Ten ekran umożliwia zainicjowanie nowej transakcji transferu środków. Po lewej stronie znajdują się dwa pola wyboru: **From Account** i **To Account**, reprezentujące konto źródłowe i docelowe przelewu. Pola te są realizowane jako rozwijane listy (`ComboBox`) zawierające numery kont i nazwiska właścicieli. Lista ta jest wypełniana przez `TransfersViewModel` z

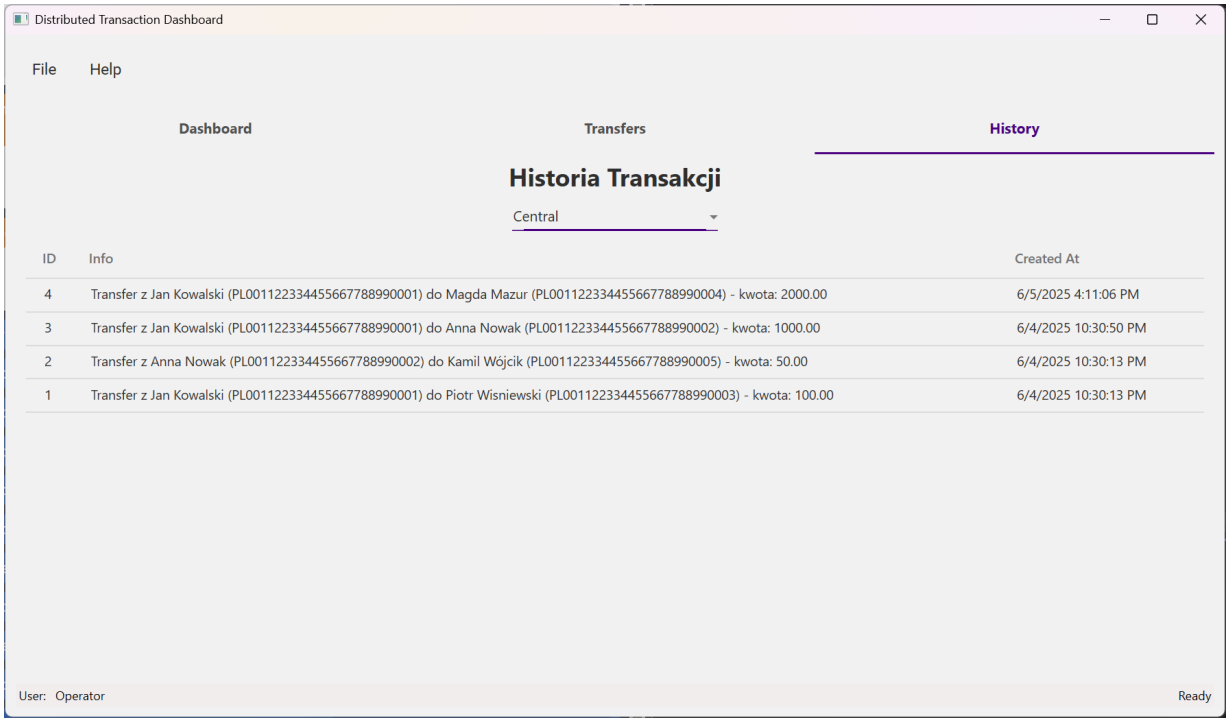
wykorzystaniem metody `accountRepo.GetAll()` – dzięki czemu zawiera wszystkie konta z centrali (co stanowi globalną listę). Dodatkowo obok każdego pola znajduje się przycisk "**...**" (szukaj), który otwiera okno dialogowe wyszukiwania klienta. Po jego kliknięciu `TransfersViewModel` wywołuje metodę `PickClientAndAccount`, która:

1. Uruchamia okno **SearchCustomerView** (dialog modalny) z podpiętym **SearchCustomerViewModel**.
2. Użytkownik w dialogu może wpisać fragment nazwiska, a lista klientów poniżej będzie się filtrować (metoda `SearchCustomerViewModel.FilterText` przy każdym ustawieniu wartości wywołuje `LoadCustomers()`, która pobiera z `ICustomerRepository.GetAll(filter)` listę pasujących osób).
3. Po wybraniu klienta i zatwierdzeniu (przycisk **OK**) dialog zostaje zamknięty, a `PickClientAndAccount` otrzymuje `vm.SelectedCustomer`. Następnie na podstawie `CustomerId` znajduje powiązane konto w kolekcji `Accounts` (wcześniej załadowanej) i ustawia je jako `SelectedFromAccount` lub `SelectedToAccount` (zależnie od tego, który przycisk wywołał dialog).

W ten sposób użytkownik może szybko wyszukać rachunek po nazwisku zamiast przewijać całą listę kont. Po wybraniu kont, w środkowej części formularza wprowadza się kwotę (pole **Amount**) oraz opcjonalny opis (**Description**) transakcji. Przy wprowadzaniu kwoty ViewModel sprawdza poprawność (czy liczba, czy dodatnia itp.) i udostępnia właściwość `CanExecuteTransfer`, która determinuje dostępność przycisków akcji.

Na dole znajduje się jeden przycisk: **Wykonaj**. Uruchamia normalny przebieg transakcji – po kliknięciu `ExecuteTransferCommand` wywołuje w ViewModelu metodę `Execute(false)`, która z kolei używa `TransferService.ExecuteDistributedTransfer(selectedFrom, selectedTo, amount, simulateError:false)`. W wyniku tego wywołania następuje wywołanie logiki omówionej w rozdz. 4 (procedura `sp_TransferDistributed` itp.). Jeśli operacja się powiedzie, ViewModel odświeża listę kont (`LoadAccounts()`) – dzięki czemu na liście widoczne są zaktualizowane salda kont od razu po przelewie. W przeciwnym razie, jeśli pojawi się wyjątek (np. brak środków, błąd połączenia), zostanie on przechwycony i komunikat błędu zostanie pokazany w interfejsie (`ErrorMessage` jest zbindowane do kontrolki tekstowej czerwonym tekstem poniżej przycisków).

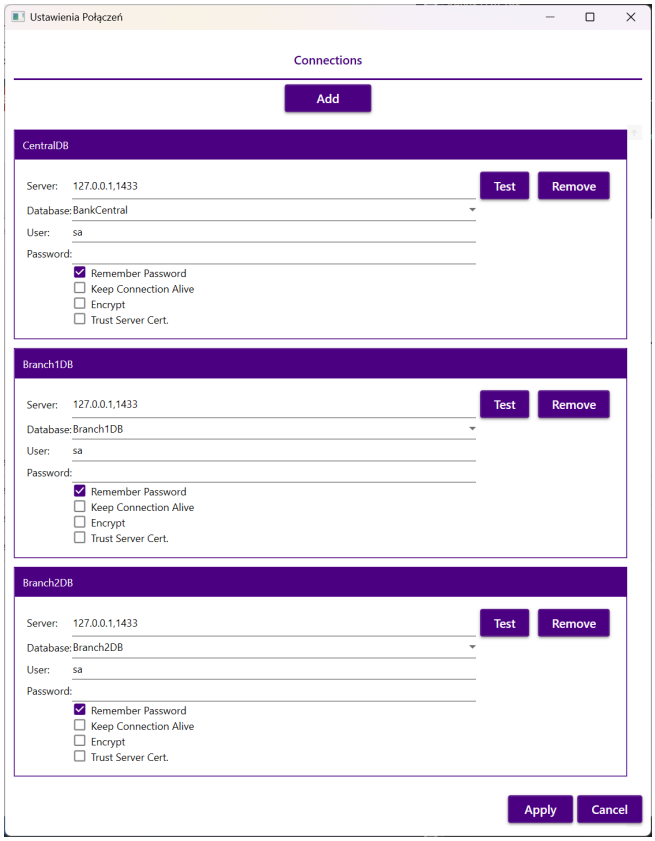
Na górze formularza Transfers widoczne jest pole **Error Message**, które pozostaje puste przy powodzeniu operacji, a wypełnia się komunikatem w razie nieudanej próby. Np. jeśli użytkownik spróbuje przelać kwotę większą niż dostępna, pojawi się komunikat *"Niewystarczające środki w oddziale."* (zgodnie z wyjątkiem rzucanym w `SqlBranchClientRepository.Debit`) lub *"Brak wystarczających środków"* (jeśli błąd wykryje procedura SQL centrali). Dzięki temu użytkownik jest informowany o przyczynie niepowodzenia transakcji.



Rysunek 4: Widok zakładki **History**.

Ta zakładka służy do przeglądania logów operacji zapisanych w systemie. U góry znajduje się przełącznik wyboru źródła historii – w implementacji jest to **ComboBox** lub **RadioButtony** pozwalające wybrać `Central` (historia centrali) lub konkretny oddział (Branch 1, Branch 2). `HistoryViewModel.SelectedBranch` jest powiązany z tym kontrolką i przy zmianie wartości automatycznie wywołuje metodę `LoadHistoryAsync()`, która korzysta z `HistoryQueryService.GetBranchHistoryAsync(selectedBranch)`. W efekcie, przy wybraniu np. „Oddział 1", aplikacja pobierze wszystkie wpisy z `Branch1DB.dbo.History` i załaduje je do listy `Entries`. Lista ta jest wyświetlana centralnie jako tabela (ListView lub DataGrid) z kolumnami: **Timestamp** (data/godzina) oraz **Info** (treść logu). Na przykład, na zrzucie widać wpisy takie jak „Synchronizacja

kont lokalnych zakończona sukcesem" (który został dodany do historii oddziału podczas inicjalizacji bazy) czy „Dodano nowe transakcje klientów z centrali" (również wpis z inicjalizacji). Gdyby wykonano przelewy podczas działania aplikacji, pojawiłyby się tutaj też logi transferów rozproszonych – w centrali byłyby to wpisy typu *"Transfer z Oddziału X do Oddziału Y..."* generowane w `sp_TransferDistributed`, w historii oddziałów natomiast obecnie manualnie dodajemy tylko wpisy testowe (operacje finansowe oddziału są logowane raczej w `BranchTransactions`). Zakładka Historia pozwala zatem monitorować pewne zdarzenia systemowe. Implementacja jest asynchroniczna (wykorzystano `async/await`), aby nie blokować interfejsu podczas zapytań do bazy.



Rysunek 5: Widok zakładki **Settings**.

Tutaj użytkownik (administrator) ma możliwość zarządzania parametrami połączenia z bazami danych. Aplikacja oczekuje pliku konfiguracyjnego `appsettings.json` zawierającego sekcję `ConnectionStrings` z nazwami `CentralDB`, `Branch1DB`, `Branch2DB`. Zakładka Ustawienia wczytuje te dane przy starcie (`SettingsViewModel` w konstruktorze wywołuje `AppSettings.Load()` – deserializację JSON do obiektu `AppSettings`, a następnie tworzy kolekcję `Connections` typu `ObservableCollection<DbConnectionSetting>`). W interfejsie widzimy tabelę (grid) z wierszami reprezentującymi poszczególne connection stringi. Dla każdego wiersza dostępne są pola edycyjne: **Key** (nazwa, np. "CentralDB"), **Server**, **Database**, **UserName**, **Password** oraz opcje **RememberPassword**, **Encrypt**, **TrustServerCertificate** (te dwa dotyczą szyfrowania połączenia) i **KeepConnectionAlive** (nieużywane w tej implementacji). Użytkownik może edytować te wartości – np. zmienić serwer SQL lub hasło do bazy. Po prawej stronie znajduje się lista dostępnych baz danych (**Available Databases**) dla aktualnie wybranego połączenia – jest ona automatycznie ładowana, gdy użytkownik zmieni pole **Server** lub wybierze inny wiersz. Mechanizm jest następujący: `SelectedConnection` ma `setter`, który wywołuje metodę `LoadDatabasesForConnection(c)`. Ta metoda tworzy tymczasowy connection string do bazy systemowej `master` na podanym serwerze i loguje się podanym loginem – następnie wykonuje zapytanie `SELECT name FROM sys.databases`, aby pobrać listę wszystkich baz. Wyniki dodaje do kolekcji `AvailableDatabases`. Dzięki temu użytkownik może rozwinąć listę i wybrać docelową nazwę bazy (np. "BankCentral"lub inną), co ogranicza ryzyko literówki. Gdy użytkownik wybrał bazę z listy, `DbConnectionSetting.Database` zostaje ustawione. Proces ten jest powtarzany dla każdej definicji połączenia.

Po wprowadzeniu zmian dostępne są na dole dwa przyciski: **Test Connection** oraz **Apply**. **Test Connection** wywołuje asynchronicznie metodę `TestConnectionAsync(c)`, która próbuje otworzyć rzeczywiste połączenie do wskazanej bazy (składając z `DbConnectionSetting` pełny connection string, podobnie jak `Apply`) i ewentualnie wyświetla komunikat, czy połączenie się powiodło, czy nie (błąd autoryzacji/serwer nieosiągalny itp. – komunikat w oknie dialogowym). **Apply** natomiast zapisuje aktualny stan ustawień do pliku. Realizuje to `SettingsViewModel.ApplyCommand`, który zbiera wszystkie `DbConnectionSetting` z listy do słownika i wywołuje `AppSettings.Save(path)` – ta metoda serializuje obiekt do JSON i nadpisuje plik. Aplikacja wyświetla komunikat "Ustawienia zapisane."dla potwierdzenia. Zastosowane podejście pozwala dynamicznie zmieniać np. na jakim serwerze działają bazy demo (domyślnie zakładamy lokalny serwer lub nazwany), jakie hasła mają loginy, itp., bez rekompilacji aplikacji. Przy kolejnym uruchomieniu aplikacja będzie używać zaktualizowanych connection stringów.

Warto wspomnieć, że w implementacji UI zastosowano framework **Material Design in XAML Toolkit**, który dostarcza gotowe style i kontrolki zgodne z wytycznymi Material Design. Dzięki temu aplikacja ma nowoczesny wygląd – np. przyciski, pola tekstowe i menu są ostylowane, a okno dialogowe wyszukiwania klienta korzysta z `MaterialDesignDialogHost` zapewniającego efekt wyciemnienia tła i animowane pojawienie się okna. Dodatkowo użyto kontrolki `DataGrid` z dynamicznym filtrowaniem (wyszukiwanie klienta) oraz mechanizmu `RelayCommand` do obsługi zdarzeń (klasyczny wzorzec delegowania komend w MVVM). Całość tworzy spójny interfejs.

Podsumowując, aplikacja WPF umożliwia wygodne przetestowanie funkcjonalności biblioteki w warunkach zbliżonych do rzeczywistych:

- Administrator może obserwować statystyki i podział transakcji między oddziałami (np. by stwierdzić, który oddział generuje więcej ruchu).
- Może wykonywać przelewy między dowolnymi kontami (wykorzystując mechanizm transakcji rozproszonych "pod maską").
- Może śledzić historię zdarzeń systemowych i operacji (przydatne do audytu).
- Może dostosować parametry połączeń – co ułatwia wdrożenie aplikacji w innym środowisku SQL Server.

Aplikacja ta pełni rolę demonstracyjną – w środowisku produkcyjnym normalnie użytkownicy korzystaliby z dedykowanego interfejsu np. w aplikacji klienckiej banku, a transakcje rozproszone zachodziłyby za kulisami. Jednak stworzony panel administracyjny doskonale pokazuje integrację wszystkich komponentów projektu.

7 Podsumowanie i wnioski

Projekt „Przetwarzanie Rozproszone w SQL Server" zrealizował cel, jakim było zaprezentowanie działania rozproszonych transakcji 2PC w środowisku bazodanowym Microsoft SQL Server, w tym z użyciem mechanizmu MSDTC. Osiągnięte zostały następujące rezultaty:

- Zaprojektowano i zaimplementowano model danych obejmujący centralny system bankowy z oddziałami, z podziałem na dane globalne (centrala) i lokalne (oddziały). Struktura ta uwzględnia realne zagadnienia, takie jak synchronizacja danych pomiędzy centralą a oddziałami.
- Utworzono procedury składowane `sp_TransferCentral` i `sp_TransferDistributed`, realizujące odpowiednio lokalny i rozproszony transfer środków, z pełnym zachowaniem integralności transakcyjnej. Zwłaszcza `sp_TransferDistributed` demonstruje wykorzystanie protokołu dwufazowego zatwierdzania – testy wykazały poprawność jej działania zarówno w warunkach normalnych, jak i przy wymuszonym błędzie.
- Napisano warstwę dostępu do danych w C# (bibliotekę API) wykorzystującą Dapper do wydajnej komunikacji z bazą. Kod jest czysty i prosty w użyciu dla warstwy wyższej – np. wywołanie `TransferDistributed` z punktu widzenia programisty wygląda identycznie jak każda inna metoda, mimo że wewnątrz angażuje skomplikowaną logikę rozproszoną.
- Przygotowano wszechstronne testy jednostkowe, które automatycznie weryfikują wszystkie kluczowe funkcje systemu. Testy te stanowią dowód poprawności implementacji i mogą służyć jako regresyjne – w przyszłości, po modyfikacjach kodu, ich ponowne uruchomienie upewni, że nie wprowadzono regresji.
- Opracowano aplikację kliencką z interfejsem graficznym, która integruje całość: umożliwia wykonywanie operacji i obserwację wyników w czasie rzeczywistym (np. po zrobieniu przelewu od razu widać zmiany sald na liście kont, pojawia się wpis w historii, a dashboard mógłby zaktualizować statystyki transakcji).

Dzięki temu projekt może służyć jako **demonstrator technologii** – pokazuje praktyczne użycie MSDTC i 2PC na przykładzie zrozumiałym (system bankowy z przelewami). Może być pomocny dla programistów chcących nauczyć się, jak od strony kodu klienckiego i bazy skonfigurować i używać transakcji rozproszonych.

Naturalnie, w ramach dalszego rozwoju i usprawniania rozwiązania można zaproponować kilka kierunków:

- **Obsługa wielu kont per klient / wiele oddziałów**: Obecna implementacja dla uproszczenia zakłada 1 konto per klient i dwa oddziały. Można rozszerzyć system na dowolną liczbę oddziałów (od strony bazy centralnej jest to łatwe – tabela `Branches` może mieć więcej wpisów; trudniej byłoby utrzymywać wiele baz oddziałowych, ale to kwestia skryptów). Wsparcie wielu kont na klienta wymagałoby doprecyzowania struktury `BranchClients` (oddział musiałby rozróżniać różne konta tego samego klienta – np. dodając kolumnę AccountId lub zmieniając `CentralClientId` na `CentralAccountId`). Wiązałoby się to z modyfikacją procedur `sp_TransferDistributed`, `sp_TransferCentral` oraz kodu UI (wybór konkretnego konta, nie tylko klienta).
- **Użycie mechanizmu `TransactionScope`** w .NET: Jak wspomniano, zamiast korzystać z procedury `sp_TransferDistributed`, można by zaimplementować logikę transferu rozproszonego w kodzie C#, otwierając dwa połączenia (do centrali i oddziału) wewnątrz jednego `TransactionScope`. Byłoby to nieco mniej wydajne (dwa roundtripy zamiast jednego wywołania SP), ale bardziej elastyczne pod kątem obsługi błędów specyficznych (można np. próbować ponawiać część operacji). Projekt mógłby porównać te dwa podejścia.

- **Zabezpieczenia i spójność danych**: W środowisku produkcyjnym należałoby zadbać o to, by np. po awarii jednego z oddziałów jego dane zostały zsynchronizowane z centralą (w naszym modelu zakładamy, że centrala dysponuje pełną wiedzą, ale salda lokalne muszą być spójne – tu `LastSync` mógłby służyć wykrywaniu różnic). Można by zaimplementować usługę synchronizacji okresowej: centrala pobiera od oddziałów różnice lub odwrotnie. W projekcie demonstracyjnym nie było to konieczne.
- **Interfejs użytkownika dla klientów**: Aplikacja stworzona jest dla administratora; rozszerzeniem mogłaby być aplikacja dla klientów banku (np. umożliwiająca zalogowanie się klientowi i wykonanie przelewu z jego konta – w tle to ten sam mechanizm, ale UI musi być inny, uwierzytelnianie itp.). Niemniej sam silnik transakcyjny mógłby zostać wykorzystany bez zmian w takiej aplikacji.
- **Szersze testy integracyjne**: Można by rozważyć napisanie skryptów T-SQL testujących procedury `sp_TransferDistributed` i `sp_TransferCentral` bezpośrednio po stronie SQL (nie tylko poprzez aplikację). To pozwoliłoby sprawdzić np. zachowanie w scenariuszach z uszkodzeniem połączenia (symulując np. spóźnioną odpowiedź jednego z uczestników). W projekcie zrealizowano testy głównie od strony aplikacji, co już jest przekrojowe, ale testy czysto bazodanowe mogłyby dać dodatkową pewność.

Reasumując, projekt można uznać za udany – wszystkie zaplanowane funkcjonalności zostały zaimplementowane i zweryfikowane. System zapewnia zachowanie spójności danych finansowych w warunkach rozproszenia geograficznego (oddzielne bazy dla oddziałów) poprzez zastosowanie protokołu 2PC koordynowanego przez MSDTC. Dokumentacja ta, wraz z dołączonym kodem źródłowym i skryptami bazodanowymi, może służyć jako pomoc dydaktyczna lub punkt wyjścia do budowy bardziej zaawansowanych systemów transakcyjnych.

Załączniki

A. Kod warstwy dostępu do danych (TransDemo.Data.Repositories)

```
 using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TransDemo.Data.Repositories
{
    /// <summary>
    /// Interface for distributed transfer operations between accounts.
    /// </summary>
    public interface IDistributedTransferRepository
    {
        /// <summary>
        /// Transfers a specified amount from one account to another in a distributed manner.
        /// </summary>
        /// <param name="fromAccId">The ID of the source account.</param>
        /// <param name="toAccId">The ID of the destination account.</param>
        /// <param name="amount">The amount to transfer.</param>
        void TransferDistributed(int fromAccId, int toAccId, decimal amount);
    }
}
```

Listing 3: IDistributedTransferRepository.cs

```
 using   Microsoft.Data.SqlClient;
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Dapper;


namespace TransDemo.Data.Repositories
{
    /// <summary>
    /// Repository for handling distributed transfer operations using SQL Server.
    /// </summary>
    public class SqlDistributedTransferRepository : IDistributedTransferRepository
    {
        private readonly string _connCentral;

        /// <summary>
```

```
        /// Initializes a new instance of the <see cref="SqlDistributedTransferRepository"/> class.
        /// </summary>
        /// <param name="connCentral">The connection string to the central database.</param>
        public SqlDistributedTransferRepository(string connCentral) => _connCentral = connCentral;

        /// <summary>
        /// Executes a distributed transfer between two accounts by calling a stored procedure.
        /// </summary>
        /// <param name="fromAccId">The ID of the account to transfer from.</param>
        /// <param name="toAccId">The ID of the account to transfer to.</param>
        /// <param name="amount">The amount to transfer.</param>
        public void TransferDistributed(int fromAccId, int toAccId, decimal amount)
        {
            // Create and open a new SQL connection using the provided connection string.
            using IDbConnection conn = new SqlConnection(_connCentral);
            conn.Open();

            // Execute the stored procedure for distributed transfer with the specified parameters.
            conn.Execute(
                "EXEC dbo.sp_TransferDistributed @FromAccId, @ToAccId, @Amount",
                new { FromAccId = fromAccId, ToAccId = toAccId, Amount = amount });
        }
    }
}
```

Listing 4: SqlDistributedTransferRepository.cs

```
   // TransDemo.Data/Repositories/ICentralBankRepository.cs
using System;

namespace TransDemo.Data.Repositories
{
    public interface ICentralBankRepository
    {
        /// <summary>
        /// Przekazuje środki między rachunkami centralnymi za pomocą sp_TransferCentral.
        /// Rzuca wyjątek, jeśli np. brak środków.
        /// </summary>
        void TransferCentral(int fromAccId, int toAccId, decimal amount);
    }
}
```

Listing 5: ICentralBankRepository.cs

```
   // TransDemo.Data/Repositories/SqlCentralBankRepository.cs
using System.Data;
using Dapper;
using Microsoft.Data.SqlClient;

namespace TransDemo.Data.Repositories
{
    /// <summary>
    /// Repository for central bank operations using SQL Server.
    /// </summary>
    public class SqlCentralBankRepository : ICentralBankRepository
    {
        private readonly string _connCentral;

        /// <summary>
        /// Initializes a new instance of the <see cref="SqlCentralBankRepository"/> class.
        /// </summary>
        /// <param name="connCentral">The connection string to the central bank database.</param>
        public SqlCentralBankRepository(string connCentral) => _connCentral = connCentral;

        /// <summary>
        /// Transfers funds between central accounts using the stored procedure sp_TransferCentral.
        /// Throws an exception if, for example, there are insufficient funds.
        /// </summary>
        /// <param name="fromAccId">Source account ID.</param>
        /// <param name="toAccId">Destination account ID.</param>
        /// <param name="amount">Amount to transfer.</param>
        public void TransferCentral(int fromAccId, int toAccId, decimal amount)
        {
            // Create and open a new SQL connection using the provided connection string.
            using var conn = new SqlConnection(_connCentral);
            conn.Open();

            // Execute the stored procedure to perform the transfer.
            conn.Execute(
```

```
                "EXEC dbo.sp_TransferCentral @FromAccId, @ToAccId, @Amount",
                new { FromAccId = fromAccId, ToAccId = toAccId, Amount = amount });
        }
    }
}
```

Listing 6: SqlCentralBankRepository.cs

```
 using   System.Collections.Generic;
using TransDemo.Models;

namespace TransDemo.Data.Repositories
{
    /// <summary>
    /// Defines methods for accessing and modifying account data.
    /// </summary>
    public interface IAccountRepository
    {
        /// <summary>
        /// Retrieves all accounts.
        /// </summary>
        /// <returns>An enumerable collection of <see cref="Account"/> objects.</returns>
        IEnumerable<Account> GetAll();

        /// <summary>
        /// Debits the specified amount from the account with the given ID.
        /// </summary>
        /// <param name="accountId">The unique identifier of the account.</param>
        /// <param name="amount">The amount to debit.</param>
        void Debit(int accountId, decimal amount);

        /// <summary>
        /// Credits the specified amount to the account with the given ID.
        /// </summary>
        /// <param name="accountId">The unique identifier of the account.</param>
        /// <param name="amount">The amount to credit.</param>
        void Credit(int accountId, decimal amount);
    }
}
```

Listing 7: IAccountRepository.cs

```
 using   System.Collections.Generic;
using System.Data;
using Dapper;
using Microsoft.Data.SqlClient;
using TransDemo.Models;

namespace TransDemo.Data.Repositories
{
    /// <summary>
    /// Provides SQL Server-based implementation of IAccountRepository for managing accounts.
    /// </summary>
    public class SqlAccountRepository : IAccountRepository
    {
        private readonly string _connString;

        /// <summary>
        /// Initializes a new instance of the <see cref="SqlAccountRepository"/> class.
        /// </summary>
        /// <param name="connString">The connection string to the SQL Server database.</param>
        public SqlAccountRepository(string connString) => _connString = connString;

        /// <summary>
        /// Creates and returns a new SQL database connection.
        /// </summary>
        /// <returns>An <see cref="IDbConnection"/> instance.</returns>
        private IDbConnection Connection() => new SqlConnection(_connString);

        /// <summary>
        /// Retrieves all accounts from the database.
        /// </summary>
        /// <returns>An enumerable collection of <see cref="Account"/> objects.</returns>
        public IEnumerable<Account> GetAll()
        {
            using var db = Connection();
            db.Open();
            // Retrieve all accounts with their details from the Accounts table
            return db.Query<Account>(
```

```
                @"SELECT
                AccountId,
                AccountNumber,
                Balance,
                CustomerId,
                BranchId
             FROM dbo.Accounts");
        }

        /// <summary>
        /// Debits the specified amount from the account with the given ID.
        /// </summary>
        /// <param name="accountId">The unique identifier of the account.</param>
        /// <param name="amount">The amount to debit.</param>
        public void Debit(int accountId, decimal amount)
        {
            using var db = Connection();
            db.Open();
            // Subtract the specified amount from the account's balance
            db.Execute("UPDATE Accounts SET Balance = Balance - @amt WHERE AccountId = @id",
                       new { amt = amount, id = accountId });
        }

        /// <summary>
        /// Credits the specified amount to the account with the given ID.
        /// </summary>
        /// <param name="accountId">The unique identifier of the account.</param>
        /// <param name="amount">The amount to credit.</param>
        public void Credit(int accountId, decimal amount)
        {
            using var db = Connection();
            db.Open();
            // Add the specified amount to the account's balance
            db.Execute("UPDATE Accounts SET Balance = Balance + @amt WHERE AccountId = @id",
                       new { amt = amount, id = accountId });
        }
    }
}
```

Listing 8: SqlAccountRepository.cs

```
   // TransDemo.Data/Repositories/IBranchClientRepository.cs
namespace TransDemo.Data.Repositories
{
    public interface IBranchClientRepository
    {
        /// <summary>
        /// Pobiera stan lokalnego konta danego klienta (wg CentralClientId).
        /// </summary>
        decimal GetBalance(int centralClientId);

        /// <summary>
        /// Obciąża lokalne konto klienta i zapisuje rekord w BranchTransactions.
        /// Rzuca, jeśli niewystarczające środki.
        /// </summary>
        void Debit(int centralClientId, decimal amount, string description);

        /// <summary>
        /// Zasila lokalne konto klienta i zapisuje rekord w BranchTransactions.
        /// </summary>
        void Credit(int centralClientId, decimal amount, string description);
    }
}
```

Listing 9: IBranchClientRepository.cs

```
   // TransDemo.Data/Repositories/SqlBranchClientRepository.cs
using System;
using System.Data;
using Dapper;
using Microsoft.Data.SqlClient;


namespace TransDemo.Data.Repositories
{
    /// <summary>
    /// Repository for managing branch client accounts and transactions using SQL Server.
    /// </summary>
    public class SqlBranchClientRepository : IBranchClientRepository
```

```
    {
        private readonly string _connString;

        /// <summary>
        /// Initializes a new instance of the <see cref="SqlBranchClientRepository"/> class.
        /// </summary>
        /// <param name="connString">The SQL Server connection string.</param>
        public SqlBranchClientRepository(string connString) => _connString = connString;

        /// <summary>
        /// Creates and returns a new SQL database connection.
        /// </summary>
        private IDbConnection Connection() => new SqlConnection(_connString);

        /// <inheritdoc />
        public decimal GetBalance(int centralClientId)
        {
            using var db = Connection();
            db.Open();
            // Retrieve the current balance for the specified central client.
            return db.QuerySingle<decimal>(
                "SELECT Balance FROM dbo.BranchClients WHERE CentralClientId = @cid",
                new { cid = centralClientId });
        }

        /// <inheritdoc />
        public void Debit(int centralClientId, decimal amount, string description)
        {
            using var db = Connection();
            db.Open();

            // 1) Check current balance
            var bal = db.QuerySingle<decimal>(
                "SELECT Balance FROM dbo.BranchClients WHERE CentralClientId = @cid",
                new { cid = centralClientId });
            if (bal < amount)
                throw new InvalidOperationException("Niewystarczające środki w oddziale.");

            // 2) Debit the account
            db.Execute(
                "UPDATE dbo.BranchClients SET Balance = Balance - @amt WHERE CentralClientId = @cid",
                new { amt = amount, cid = centralClientId });

            // 3) Record the transaction in history
            db.Execute(
                @"INSERT INTO dbo.BranchTransactions(BranchClientId, Amount, TxnType, Description)
                      SELECT BranchClientId, @amt, 'DEBIT', @desc
                      FROM dbo.BranchClients WHERE CentralClientId = @cid",
                new { amt = amount, cid = centralClientId, desc = description });
        }

        /// <inheritdoc />
        public void Credit(int centralClientId, decimal amount, string description)
        {
            using var db = Connection();
            db.Open();

            // Credit the account
            db.Execute(
                "UPDATE dbo.BranchClients SET Balance = Balance + @amt WHERE CentralClientId = @cid",
                new { amt = amount, cid = centralClientId });

            // Record the transaction in history
            db.Execute(
                @"INSERT INTO dbo.BranchTransactions(BranchClientId, Amount, TxnType, Description)
                      SELECT BranchClientId, @amt, 'CREDIT', @desc
                      FROM dbo.BranchClients WHERE CentralClientId = @cid",
                new { amt = amount, cid = centralClientId, desc = description });
        }
    }
}
```

Listing 10: SqlBranchClientRepository.cs

```
  using   System.Collections.Generic;
using TransDemo.Models;

namespace TransDemo.Data.Repositories
{
    /// <summary>
```

```
    /// Defines methods for accessing customer data.
    /// </summary>
    public interface ICustomerRepository
    {
        /// <summary>
        /// Retrieves all customers, optionally filtered by a search string.
        /// </summary>
        /// <param name="filter">An optional filter string to search customers by name or other criteria
    .</param>
        /// <returns>An enumerable collection of <see cref="Customer"/> objects.</returns>
        IEnumerable<Customer> GetAll(string? filter = null);
    }
}
```

Listing 11: ICustomerRepository.cs

```
 using   System.Collections.Generic;
using System.Data;
using Dapper;
using Microsoft.Data.SqlClient;
using TransDemo.Models;

namespace TransDemo.Data.Repositories
{
    // Repository implementation for accessing Customer data from SQL database
    public class SqlCustomerRepository : ICustomerRepository
    {
        private readonly string _connString;

        // Constructor accepting connection string
        public SqlCustomerRepository(string connString) => _connString = connString;

        // Helper method to create a new SQL connection
        private IDbConnection Connection() => new SqlConnection(_connString);

        // Retrieves all customers, optionally filtered by full name
        public IEnumerable<Customer> GetAll(string? filter = null)
        {
            using var db = Connection();
            db.Open();
            // Base SQL query to select customer ID and full name
            var baseSql = "SELECT CustomerId, FirstName + ' ' + LastName AS FullName FROM Customers";
            if (string.IsNullOrWhiteSpace(filter))
                // No filter provided, return all customers
                return db.Query<Customer>(baseSql);
            // Filter provided, return customers matching the filter
            return db.Query<Customer>(
                baseSql + " WHERE FirstName + ' ' + LastName LIKE @f",
                new { f = $"%{filter}%" });
        }
    }
}
```

Listing 12: SqlCustomerRepository.cs

```
 using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using TransDemo.Models;

namespace TransDemo.Data.Repositories
{
    /// <summary>
    /// Interface for managing history entries in different branches and the central repository.
    /// </summary>
    public interface IHistoryRepository
    {
        /// <summary>
        /// Adds a history entry to branch A.
        /// </summary>
        /// <param name="info">The information to be added as a history entry.</param>
        void AddEntryToA(string info);

        /// <summary>
        /// Adds a history entry to branch B.
        /// </summary>
        /// <param name="info">The information to be added as a history entry.</param>
```

```
        void AddEntryToB(string info);

        /// <summary>
        /// Retrieves the history entries from a specific branch.
        /// </summary>
        /// <param name="branchId">The identifier of the branch.</param>
        /// <returns>An enumerable collection of history entries from the specified branch.</returns>
        IEnumerable<HistoryEntry> GetHistoryFromBranch(int branchId);

        /// <summary>
        /// Retrieves the central history entries.
        /// </summary>
        /// <returns>An enumerable collection of central history entries.</returns>
        IEnumerable<HistoryEntry> GetCentralHistory();
    }
}
```

Listing 13: IHistoryRepository.cs

```
 using   Dapper;
using Microsoft.Data.SqlClient;
using System.Collections.Generic;
using TransDemo.Models;

namespace TransDemo.Data.Repositories
{
    /// <summary>
    /// Repository for managing history entries in central and branch databases.
    /// </summary>
    public class SqlHistoryRepository : IHistoryRepository
    {
        private readonly string _connCentral;
        private readonly string _connA;
        private readonly string _connB;

        /// <summary>
        /// Initializes a new instance of the <see cref="SqlHistoryRepository"/> class.
        /// </summary>
        /// <param name="connCentral">Connection string to the central database.</param>
        /// <param name="connA">Connection string to branch A database.</param>
        /// <param name="connB">Connection string to branch B database.</param>
        public SqlHistoryRepository(string connCentral, string connA, string connB)
        {
            _connCentral = connCentral;
            _connA = connA;
            _connB = connB;
        }

        /// <summary>
        /// Adds a new history entry to branch A database.
        /// </summary>
        /// <param name="info">The information to be added to the history.</param>
        public void AddEntryToA(string info)
        {
            using var conn = new SqlConnection(_connA);
            conn.Open();
            conn.Execute("INSERT INTO dbo.History (Info) VALUES (@info)", new { info });
        }

        /// <summary>
        /// Adds a new history entry to branch B database.
        /// </summary>
        /// <param name="info">The information to be added to the history.</param>
        public void AddEntryToB(string info)
        {
            using var conn = new SqlConnection(_connB);
            conn.Open();
            conn.Execute("INSERT INTO dbo.History (Info) VALUES (@info)", new { info });
        }

        /// <summary>
        /// Retrieves the history entries from the specified branch.
        /// </summary>
        /// <param name="branchId">The branch identifier (1 for A, otherwise B).</param>
        /// <returns>A collection of <see cref="HistoryEntry"/> from the branch database.</returns>
        public IEnumerable<HistoryEntry> GetHistoryFromBranch(int branchId)
        {
            var connStr = branchId == 1 ? _connA : _connB;
            using var conn = new SqlConnection(connStr);
            conn.Open();
```

```
            return conn.Query<HistoryEntry>(
                "SELECT HistoryId, Info, CreatedAt FROM dbo.History ORDER BY CreatedAt DESC");
        }

        /// <summary>
        /// Retrieves the history entries from the central database.
        /// </summary>
        /// <returns>A collection of <see cref="HistoryEntry"/> from the central database.</returns>
        public IEnumerable<HistoryEntry> GetCentralHistory()
        {
            using var conn = new SqlConnection(_connCentral);
            conn.Open();
            return conn.Query<HistoryEntry>(
                "SELECT HistoryId, Info, CreatedAt FROM dbo.History ORDER BY CreatedAt DESC");
        }
    }
}
```

Listing 14: SqlHistoryRepository.cs

B. Kod warstwy logiki (TransDemo.Logic/Services)

```
   // TransDemo.Logic/Services/TransferService.cs
using System;
using System.Collections.Generic;
using System.Transactions;
using TransDemo.Data.Repositories;
using TransDemo.Models;

namespace TransDemo.Logic.Services
{
    /// <summary>
    /// Service responsible for handling distributed transfer operations between accounts.
    /// </summary>
    public class TransferService
    {
        private readonly IDistributedTransferRepository _distributed;

        /// <summary>
        /// Initializes a new instance of the <see cref="TransferService"/> class.
        /// </summary>
        /// <param name="distributed">The distributed transfer repository used to perform transfer
    operations.</param>
        public TransferService(IDistributedTransferRepository distributed)
        {
            _distributed = distributed;
        }

        /// <summary>
        /// Executes a distributed transfer between two accounts.
        /// </summary>
        /// <param name="from">The source account from which the amount will be debited.</param>
        /// <param name="to">The destination account to which the amount will be credited.</param>
        /// <param name="amount">The amount to transfer.</param>
        /// <param name="simulateError">If set to <c>true</c>, simulates an error by throwing an
    exception.</param>
        /// <exception cref="InvalidOperationException">Thrown when <paramref name="simulateError"/> is <
    c>true</c>.</exception>
        public void ExecuteDistributedTransfer(Account from, Account to, decimal amount, bool
    simulateError = false)
        {
            if (simulateError)
                throw new InvalidOperationException("Symulowany błąd.");
            // TransactionScope could be used here, but since the entire operation is handled in a stored
     procedure, it is not necessary.
            _distributed.TransferDistributed(from.AccountId, to.AccountId, amount);
        }
    }

}
```

Listing 15: TransferService.cs

```
  using   System.Collections.Generic;
using System.Threading.Tasks;
using Dapper;
using Microsoft.Data.SqlClient;
using Microsoft.Extensions.Configuration;
using TransDemo.Models;
```

```
7
8  namespace TransDemo.Logic.Services
9  {
10     /// <summary>
11     /// Service for retrieving transaction history from the database for individual branches.
12     /// </summary>
13     public class HistoryQueryService
14     {
15         /// <summary>
16         /// Connection string for the central database.
17         /// </summary>
18         private readonly string _connCentral;
19
20         /// <summary>
21         /// Connection string for branch 1 database.
22         /// </summary>
23         private readonly string _connB1;
24
25         /// <summary>
26         /// Connection string for branch 2 database.
27         /// </summary>
28         private readonly string _connB2;
29
30         /// <summary>
31         /// Initializes a new instance of the <see cref="HistoryQueryService"/> class.
32         /// Retrieves connection strings from the provided configuration.
33         /// </summary>
34         /// <param name="config">Configuration object from appsettings.json.</param>
35         public HistoryQueryService(IConfiguration config)
36         {
37             _connCentral = config.GetConnectionString("CentralDB")!;
38             _connB1 = config.GetConnectionString("Branch1DB")!;
39             _connB2 = config.GetConnectionString("Branch2DB")!;
40         }
41
42         /// <summary>
43         /// Retrieves the transaction history for the specified branch.
44         /// </summary>
45         /// <param name="branchNumber">
46         /// Branch number:
47         /// <list type="bullet">
48         /// <item>
49         /// <description>0 - Central database</description>
50         /// </item>
51         /// <item>
52         /// <description>1 - Branch 1</description>
53         /// </item>
54         /// <item>
55         /// <description>2 - Branch 2</description>
56         /// </item>
57         /// </list>
58         /// </param>
59         /// <returns>
60         /// A task that represents the asynchronous operation. The task result contains a list of <see
       cref="HistoryEntry"/> objects.
61         /// </returns>
62         /// <exception cref="ArgumentOutOfRangeException">Thrown when an unknown branch number is
       provided.</exception>
63         public async Task<IEnumerable<HistoryEntry>> GetBranchHistoryAsync(int branchNumber)
64         {
65             // Select the appropriate connection string based on the branch number.
66             string connStr = branchNumber switch
67             {
68                 0 => _connCentral,
69                 1 => _connB1,
70                 2 => _connB2,
71                 _ => throw new ArgumentOutOfRangeException(nameof(branchNumber), "Unknown branch number")
72             };
73
74             // Open a new SQL connection using the selected connection string.
75             await using var conn = new SqlConnection(connStr);
76             await conn.OpenAsync();
77
78             // SQL query to retrieve history entries ordered by creation date descending.
79             const string sql = @"SELECT HistoryId, Info, CreatedAt FROM History ORDER BY CreatedAt DESC";
80
81             // Execute the query and return the results as a list of HistoryEntry objects.
82             return await conn.QueryAsync<HistoryEntry>(sql);
83         }
```

```
    }
}
```

Listing 16: HistoryQueryService.cs

```
 using   Dapper;
using Microsoft.Data.SqlClient;
using Microsoft.Extensions.Configuration;
using System.Collections.Generic;
using System.Threading.Tasks;
using TransDemo.Models;

namespace TransDemo.Logic.Services
{
    /// <summary>
    /// Provides methods for retrieving dashboard statistics such as daily transactions, balances, branch shares, and top customers.
    /// </summary>
    public class DashboardStatsService
    {
        private readonly string _connCentral;

        /// <summary>
        /// Initializes a new instance of the <see cref="DashboardStatsService"/> class.
        /// </summary>
        /// <param name="config">The application configuration used to retrieve connection strings.</param>
        public DashboardStatsService(IConfiguration config)
        {
            _connCentral = config.GetConnectionString("CentralDB")!;
        }

        /// <summary>
        /// Retrieves daily transaction statistics for the specified number of most recent days.
        /// </summary>
        /// <param name="lastDays">The number of recent days to include in the statistics.</param>
        /// <returns>A collection of <see cref="DailyTransactionStat"/> objects representing daily transaction counts and sums.</returns>
        public async Task<IEnumerable<DailyTransactionStat>> GetStatsAsync(int lastDays)
        {
            var sql = @"
                    SELECT TOP (@Limit) *
                    FROM dbo.vDailyTransactionStats
                    ORDER BY TxnDate DESC";

            await using var conn = new SqlConnection(_connCentral);
            return await conn.QueryAsync<DailyTransactionStat>(sql, new { Limit = lastDays });
        }

        /// <summary>
        /// Retrieves daily cumulative balance statistics for the specified number of most recent days.
        /// </summary>
        /// <param name="lastDays">The number of recent days to include in the balance statistics.</param>
        /// <returns>A collection of <see cref="DailyBalanceStat"/> objects representing daily cumulative balances.</returns>
        public async Task<IEnumerable<DailyBalanceStat>> GetDailyBalanceAsync(int lastDays)
        {
            const string sql = @"
                SELECT TOP (@Limit) *
                FROM dbo.vDailyBalances
                ORDER BY BalanceDate DESC";

            await using var conn = new SqlConnection(_connCentral);
            return await conn.QueryAsync<DailyBalanceStat>(sql, new { Limit = lastDays });
        }

        /// <summary>
        /// Retrieves the transaction share percentage for each branch.
        /// </summary>
        /// <returns>A collection of <see cref="BranchTxnShare"/> objects representing branch transaction shares.</returns>
        public async Task<IEnumerable<BranchTxnShare>> GetBranchShareAsync()
        {
            const string sql = "SELECT * FROM dbo.vBranchTransactionShares";

            await using var conn = new SqlConnection(_connCentral);
            return await conn.QueryAsync<BranchTxnShare>(sql);
        }

```

```
        /// <summary>
        /// Retrieves the top customers by total transaction amount.
        /// </summary>
        /// <param name="top">The number of top customers to retrieve. Defaults to 5.</param>
        /// <returns>A collection of <see cref="TopCustomerStat"/> objects representing the top customers.</returns>
        public async Task<IEnumerable<TopCustomerStat>> GetTopCustomersAsync(int top = 5)
        {
            const string sql = @"
                SELECT TOP (@TopN) *
                FROM dbo.vTopCustomerTransactions
                ORDER BY TotalAmount DESC";

            await using var conn = new SqlConnection(_connCentral);
            return await conn.QueryAsync<TopCustomerStat>(sql, new { TopN = top });
        }
    }
}
```

Listing 17: DashboardStatsService.cs

C. Kod modeli danych (TransDemo.Models)

```
namespace TransDemo.Models
{
    /// <summary>
    /// Represents a bank account entity.
    /// </summary>
    public class Account
    {
        /// <summary>
        /// Gets or sets the unique identifier for the account.
        /// </summary>
        public int AccountId { get; set; }

        /// <summary>
        /// Gets or sets the account number.
        /// </summary>
        public string AccountNumber { get; set; } = "";

        /// <summary>
        /// Gets or sets the current balance of the account.
        /// </summary>
        public decimal Balance { get; set; }

        /// <summary>
        /// Gets or sets the identifier of the customer who owns the account.
        /// </summary>
        public int CustomerId { get; set; }

        /// <summary>
        /// Gets or sets the identifier of the branch where the account is held.
        /// </summary>
        public int BranchId { get; set; }
    }
}
```

Listing 18: Account.cs

```
namespace TransDemo.Models
{
    /// <summary>
    /// Represents a customer entity with an identifier and full name.
    /// </summary>
    public class Customer
    {
        /// <summary>
        /// Gets or sets the unique identifier for the customer.
        /// </summary>
        public int CustomerId { get; set; }

        /// <summary>
        /// Gets or sets the full name of the customer.
        /// </summary>
        public string FullName { get; set; } = "";
    }
}
```

Listing 19: Customer.cs

```
  namespace   TransDemo.Models
{
    /// <summary>
    /// Represents a single entry in the history log.
    /// </summary>
    public class HistoryEntry
    {
        /// <summary>
        /// Gets or sets the unique identifier for the history entry.
        /// </summary>
        public int HistoryId { get; set; }

        /// <summary>
        /// Gets or sets the information or description associated with this history entry.
        /// </summary>
        public string Info { get; set; } = "";

        /// <summary>
        /// Gets or sets the date and time when this history entry was created.
        /// </summary>
        public DateTime CreatedAt { get; set; }
    }
}
```

Listing 20: HistoryEntry.cs

```
  using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TransDemo.Models
{
    /// <summary>
    /// Represents daily statistics for transactions, including the date,
    /// the number of transactions, and the total transaction amount.
    /// </summary>
    public class DailyTransactionStat
    {
        /// <summary>
        /// Gets or sets the date of the transactions.
        /// </summary>
        public DateTime TxnDate { get; set; }

        /// <summary>
        /// Gets or sets the total number of transactions for the specified date.
        /// </summary>
        public int TxnCount { get; set; }

        /// <summary>
        /// Gets or sets the total amount of all transactions for the specified date.
        /// </summary>
        public decimal TotalAmount { get; set; }
    }
}
```

Listing 21: DailyTransactionStat.cs

```
  namespace   TransDemo.Models;

/// <summary>
/// Represents the daily cumulative balance statistics for a specific date.
/// </summary>
public class DailyBalanceStat
{
    /// <summary>
    /// Gets or sets the date for which the balance is calculated.
    /// </summary>
    public DateTime BalanceDate { get; set; }

    /// <summary>
    /// Gets or sets the cumulative balance for the specified date.
    /// </summary>
    public decimal CumulativeBalance { get; set; }
}
```

Listing 22: DailyBalanceStat.cs

```
 using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TransDemo.Models
{
    /// <summary>
    /// Represents the share percentage of a transaction for a specific branch.
    /// </summary>
    public class BranchTxnShare
    {
        /// <summary>
        /// Gets or sets the name of the branch.
        /// </summary>
        public string BranchName { get; set; } = null!;

        /// <summary>
        /// Gets or sets the share percentage of the transaction for the branch.
        /// </summary>
        public decimal SharePercent { get; set; }
    }
}
```

Listing 23: BranchTxnShare.cs

```
 using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

//
namespace TransDemo.Models
{
    /// <summary>
    /// Represents statistics for a top customer, including their full name and total transaction amount.
    /// </summary>
    public class TopCustomerStat
    {
        /// <summary>
        /// Gets or sets the full name of the customer.
        /// </summary>
        public string FullName { get; set; } = null!;

        /// <summary>
        /// Gets or sets the total transaction amount for the customer.
        /// </summary>
        public decimal TotalAmount { get; set; }
    }
}
```

Listing 24: TopCustomerStat.cs

D. Kod warstwy UI – modele i widoki

TransDemo.UI.Models:

```
 using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using System.IO;
using System.Text.Json;

namespace TransDemo.UI.Models
{
    /// <summary>
    /// Represents application settings, including connection strings.
    /// Provides methods for loading from and saving to a JSON file.
    /// </summary>
    public class AppSettings
    {
        /// <summary>
        /// Gets or sets the collection of connection strings.
        /// The key is the name of the connection, and the value is the connection string.
        /// </summary>
```

```
        public required Dictionary<string, string> ConnectionStrings { get; set; }

        /// <summary>
        /// Loads the <see cref="AppSettings"/> from a JSON file at the specified path.
        /// </summary>
        /// <param name="path">The file path to load the settings from.</param>
        /// <returns>An instance of <see cref="AppSettings"/> deserialized from the file.</returns>
        /// <exception cref="FileNotFoundException">Thrown if the file does not exist.</exception>
        /// <exception cref="JsonException">Thrown if the file content is not valid JSON.</exception>
        public static AppSettings Load(string path)
            => JsonSerializer.Deserialize<AppSettings>(File.ReadAllText(path))!;

        /// <summary>
        /// Saves the current <see cref="AppSettings"/> instance to a JSON file at the specified path.
        /// </summary>
        /// <param name="path">The file path to save the settings to.</param>
        public void Save(string path)
            => File.WriteAllText(path,
                JsonSerializer.Serialize(this, new JsonSerializerOptions { WriteIndented = true }));
    }
}
```

Listing 25: AppSettings.cs

```
 using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TransDemo.UI.Models
{
    /// <summary>
    /// Represents the settings required to establish a database connection.
    /// </summary>
    public class DbConnectionSetting
    {
        /// <summary>
        /// Gets or sets the key or name of the connection (e.g., "CentralDB", "Branch1DB").
        /// </summary>
        public string Key { get; set; }

        /// <summary>
        /// Gets or sets the server address or name where the database is hosted.
        /// </summary>
        public string Server { get; set; }

        /// <summary>
        /// Gets or sets the name of the database to connect to.
        /// </summary>
        public string Database { get; set; }

        /// <summary>
        /// Gets or sets the username used for database authentication.
        /// </summary>
        public string UserName { get; set; }

        /// <summary>
        /// Gets or sets the password used for database authentication.
        /// </summary>
        public string Password { get; set; }

        /// <summary>
        /// Gets or sets a value indicating whether the password should be remembered.
        /// </summary>
        public bool RememberPassword { get; set; }

        /// <summary>
        /// Gets or sets a value indicating whether the connection should be encrypted.
        /// </summary>
        public bool Encrypt { get; set; }

        /// <summary>
        /// Gets or sets a value indicating whether to trust the server certificate.
        /// </summary>
        public bool TrustServerCertificate { get; set; }

        /// <summary>
        /// Gets or sets a value indicating whether to keep the connection alive.
        /// </summary>
```

```
        public bool KeepConnectionAlive { get; set; }
    }
}
```

Listing 26: DbConnectionSetting.cs

TransDemo.UI.ViewModels:

```
  using   System.ComponentModel;
using System.Runtime.CompilerServices;

namespace TransDemo.UI.ViewModels
{
    /// <summary>
    /// Klasa bazowa dla wszystkich ViewModeli: udostępnia INotifyPropertyChanged
    /// </summary>
    public abstract class BaseViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler? PropertyChanged;

        /// <summary>
        /// Wywołaj, gdy zmieni się wartość właściwości
        /// </summary>
        protected void OnPropertyChanged([CallerMemberName] string? propertyName = null)
            => PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

        /// <summary>
        /// Pomocnicza metoda do ustawiania właściwości i powiadamiania UI
        /// </summary>
        protected bool SetProperty<T>(ref T field, T value, [CallerMemberName] string? propertyName =
    null)
        {
            if (Equals(field, value)) return false;
            field = value;
            OnPropertyChanged(propertyName);
            return true;
        }
    }
}
```

Listing 27: BaseViewModel.cs

```
  using   System;
using System.Windows.Input;

namespace TransDemo.UI.ViewModels
{
    /// <summary>
    /// A command whose sole purpose is to relay its functionality
    /// to other objects by invoking delegates. The default return value for the CanExecute method is '
    true'.
    /// </summary>
    public class RelayCommand : ICommand
    {
        private readonly Action<object?> _execute;
        private readonly Predicate<object?> _canExecute;

        /// <summary>
        /// Initializes a new instance of the <see cref="RelayCommand"/> class.
        /// </summary>
        /// <param name="execute">The execution logic.</param>
        /// <param name="canExecute">The execution status logic.</param>
        /// <exception cref="ArgumentNullException">Thrown if the execute argument is null.</exception>
        public RelayCommand(Action<object?> execute, Predicate<object?>? canExecute = null)
        {
            _execute = execute ?? throw new ArgumentNullException(nameof(execute));
            _canExecute = canExecute ?? (_ => true);
        }

        /// <summary>
        /// Determines whether the command can execute in its current state.
        /// </summary>
        /// <param name="parameter">Data used by the command. If the command does not require data, this
    object can be set to null.</param>
        /// <returns>true if this command can be executed; otherwise, false.</returns>
        public bool CanExecute(object? parameter) => _canExecute(parameter);

        /// <summary>
        /// Executes the command.
        /// </summary>
```

```
        /// <param name="parameter">Data used by the command. If the command does not require data, this object can be set to null.</param>
        public void Execute(object? parameter) => _execute(parameter);

        /// <summary>
        /// Occurs when changes occur that affect whether or not the command should execute.
        /// </summary>
        public event EventHandler? CanExecuteChanged
        {
            add => CommandManager.RequerySuggested += value;
            remove => CommandManager.RequerySuggested -= value;
        }
    }

    /// <summary>
    /// A generic command whose sole purpose is to relay its functionality
    /// to other objects by invoking delegates. The default return value for the CanExecute method is 'true'.
    /// </summary>
    /// <typeparam name="T">The type of the command parameter.</typeparam>
    public class RelayCommand<T> : ICommand
    {
        private readonly Action<T> _execute;
        private readonly Func<T, bool>? _canExecute;

        /// <summary>
        /// Initializes a new instance of the <see cref="RelayCommand{T}"/> class.
        /// </summary>
        /// <param name="execute">The execution logic.</param>
        /// <param name="canExecute">The execution status logic.</param>
        public RelayCommand(Action<T> execute, Func<T, bool>? canExecute = null)
        {
            _execute = execute ?? throw new ArgumentNullException(nameof(execute));
            _canExecute = canExecute;
        }

        /// <summary>
        /// Determines whether the command can execute in its current state.
        /// </summary>
        /// <param name="parameter">Data used by the command. If the command does not require data, this object can be set to null.</param>
        /// <returns>true if this command can be executed; otherwise, false.</returns>
        public bool CanExecute(object? parameter)
            => _canExecute == null || _canExecute((T)parameter!);

        /// <summary>
        /// Executes the command.
        /// </summary>
        /// <param name="parameter">Data used by the command. If the command does not require data, this object can be set to null.</param>
        public void Execute(object? parameter)
            => _execute((T)parameter!);

        /// <summary>
        /// Occurs when changes occur that affect whether or not the command should execute.
        /// </summary>
        public event EventHandler? CanExecuteChanged;

        /// <summary>
        /// Raises the <see cref="CanExecuteChanged"/> event.
        /// </summary>
        public void RaiseCanExecuteChanged()
            => CanExecuteChanged?.Invoke(this, EventArgs.Empty);
    }
}
```

Listing 28: RelayCommand.cs

```
  namespace    TransDemo.UI.ViewModels
{
    /// <summary>
    /// Abstract base ViewModel for tab items in the application.
    /// Provides properties for the tab header and the content (typically a UserControl).
    /// </summary>
    public abstract class TabItemViewModel
    {
        /// <summary>
        /// Gets the header text displayed on the tab.
        /// </summary>
        public abstract string Header { get; }
```

```
13
14          /// <summary>
15          /// Gets the content to be displayed within the tab.
16          /// Typically, this is a UserControl instance.
17          /// </summary>
18          public abstract object Content { get; }
19      }
20  }
```

Listing 29: TabItemViewModel.cs

```
1   using   TransDemo.Logic.Services;
2   using TransDemo.UI.Views;  // zakładamy, że masz DashboardView.xaml
3
4   namespace TransDemo.UI.ViewModels
5   {
6       /// <summary>
7       /// ViewModel for the Dashboard tab in the main application window.
8       /// Provides the header and content (UserControl) for the dashboard tab.
9       /// </summary>
10      public class DashboardTabViewModel : TabItemViewModel
11      {
12          /// <summary>
13          /// Gets the header text displayed on the tab.
14          /// </summary>
15          public override string Header => "Dashboard";
16
17          /// <summary>
18          /// Gets the content of the tab, which is the DashboardView UserControl.
19          /// </summary>
20          public override object Content { get; }
21
22          /// <summary>
23          /// Initializes a new instance of the <see cref="DashboardTabViewModel"/> class.
24          /// Sets the DataContext of the provided DashboardView to the given DashboardViewModel.
25          /// </summary>
26          /// <param name="view">The DashboardView UserControl to be displayed in the tab.</param>
27          /// <param name="vm">The DashboardViewModel to be used as the DataContext for the view.</param>
28          public DashboardTabViewModel(DashboardView view, DashboardViewModel vm)
29          {
30              view.DataContext = vm;
31              Content = view;
32          }
33      }
34  }
```

Listing 30: DashboardTabViewModel.cs

```
1   using   TransDemo.UI.Views;
2   using TransDemo.UI.ViewModels;
3
4   namespace TransDemo.UI.ViewModels
5   {
6       /// <summary>
7       /// ViewModel for the Transfers tab in the application.
8       /// Provides the header and content for the Transfers tab.
9       /// </summary>
10      public class TransfersTabViewModel : TabItemViewModel
11      {
12          /// <summary>
13          /// Gets the header text displayed on the Transfers tab.
14          /// </summary>
15          public override string Header => "Transfers";
16
17          /// <summary>
18          /// Gets the content to be displayed within the Transfers tab.
19          /// Typically, this is a <see cref="TransfersView"/> instance.
20          /// </summary>
21          public override object Content { get; }
22
23          /// <summary>
24          /// Initializes a new instance of the <see cref="TransfersTabViewModel"/> class.
25          /// Sets up the view and its corresponding ViewModel.
26          /// </summary>
27          /// <param name="view">The <see cref="TransfersView"/> to be displayed in the tab.</param>
28          /// <param name="vm">The <see cref="TransfersViewModel"/> to be used as the DataContext for the
    view.</param>
29          public TransfersTabViewModel(
30              TransfersView view,
```

```
            TransfersViewModel vm)
        {
            view.DataContext = vm;
            Content = view;
        }
    }
}
```

Listing 31: TransfersTabViewModel.cs

```
 using   TransDemo.Logic.Services;
using TransDemo.UI.Views;

namespace TransDemo.UI.ViewModels
{
    /// <summary>
    /// ViewModel for the History tab in the application's tab control.
    /// Provides the header and content (view) for the history tab.
    /// </summary>
    public class HistoryTabViewModel : TabItemViewModel
    {
        /// <summary>
        /// Gets the header text for the history tab.
        /// </summary>
        public override string Header => "History";

        /// <summary>
        /// Gets the content (view) associated with the history tab.
        /// </summary>
        public override object Content { get; }

        /// <summary>
        /// Initializes a new instance of the <see cref="HistoryTabViewModel"/> class.
        /// Sets up the view and binds it to the provided <see cref="HistoryViewModel"/>.
        /// </summary>
        /// <param name="vm">The view model for the history view.</param>
        public HistoryTabViewModel(HistoryViewModel vm)
        {
            var view = new HistoryView
            {
                DataContext = vm
            };
            Content = view;
        }
    }
}
```

Listing 32: HistoryTabViewModel.cs

```
 using   MaterialDesignThemes.Wpf;
using Microsoft.Extensions.DependencyInjection;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Windows.Input;
using TransDemo.Logic.Services;
using TransDemo.UI.Views;
using static System.Net.Mime.MediaTypeNames;

namespace TransDemo.UI.ViewModels
{
    /// <summary>
    /// Main ViewModel for the application. Handles tab navigation, commands, and status information.
    /// </summary>
    public class MainViewModel : INotifyPropertyChanged
    {
        /// <summary>
        /// Service provider for resolving dependencies.
        /// </summary>
        private readonly IServiceProvider _provider;

        /// <summary>
        /// Service responsible for transfer operations.
        /// </summary>
        private readonly TransferService _transferSvc;

        /// <summary>
        /// Collection of tab view models displayed in the main window.
        /// </summary>
```

```
31          public ObservableCollection<TabItemViewModel> Tabs { get; }
32
33          private TabItemViewModel _selectedTab;
34
35          /// <summary>
36          /// Gets or sets the currently selected tab.
37          /// </summary>
38          public TabItemViewModel SelectedTab
39          {
40              get => _selectedTab;
41              set => SetProperty(ref _selectedTab, value);
42          }
43
44          /// <summary>
45          /// Gets the name of the current user.
46          /// </summary>
47          public string CurrentUser { get; } = "Operator";
48
49          private string _status = "Ready";
50
51          /// <summary>
52          /// Gets the current status message.
53          /// </summary>
54          public string StatusMessage
55          {
56              get => _status;
57              private set => SetProperty(ref _status, value);
58          }
59
60          /// <summary>
61          /// Command to open the settings window.
62          /// </summary>
63          public ICommand OpenSettingsCommand { get; }
64
65          /// <summary>
66          /// Command to exit the application.
67          /// </summary>
68          public ICommand ExitCommand { get; }
69
70          /// <summary>
71          /// Command to open the about window.
72          /// </summary>
73          public ICommand OpenAboutCommand { get; }
74
75          /// <summary>
76          /// Constructor for design-time or test usage.
77          /// </summary>
78          /// <param name="svc">The transfer service.</param>
79          public MainViewModel(TransferService svc) => _transferSvc = svc;
80
81          /// <summary>
82          /// Initializes a new instance of the <see cref="MainViewModel"/> class.
83          /// </summary>
84          /// <param name="provider">The service provider for dependency resolution.</param>
85          /// <param name="svc">The transfer service.</param>
86          public MainViewModel(IServiceProvider provider, TransferService svc)
87          {
88              _provider = provider;
89              _transferSvc = svc;
90
91              OpenSettingsCommand = new RelayCommand(_ =>
92              {
93                  // Retrieve SettingsView from the service provider and show it as a dialog.
94                  var win = _provider.GetRequiredService<SettingsView>();
95                  win.Owner = System.Windows.Application.Current.MainWindow;
96                  win.ShowDialog();
97              });
98
99              ExitCommand = new RelayCommand(_ => System.Windows.Application.Current.Shutdown());
100
101             OpenAboutCommand = new RelayCommand(_ =>
102             {
103                 // TODO: Implement about window display logic.
104             });
105
106             Tabs = new ObservableCollection<TabItemViewModel>
107                 {
108                     _provider.GetRequiredService<DashboardTabViewModel>(),
109                     _provider.GetRequiredService<TransfersTabViewModel>(),
```

```
                    _provider.GetRequiredService<HistoryTabViewModel>()
                };
            SelectedTab = Tabs[0];
        }

        #region INotifyPropertyChanged

        /// <inheritdoc/>
        public event PropertyChangedEventHandler? PropertyChanged;

        /// <summary>
        /// Raises the <see cref="PropertyChanged"/> event.
        /// </summary>
        /// <param name="name">The name of the property that changed.</param>
        protected void OnPropertyChanged([CallerMemberName] string name = null!)
            => PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));

        /// <summary>
        /// Sets the property and raises <see cref="PropertyChanged"/> if the value changes.
        /// </summary>
        /// <typeparam name="T">The type of the property.</typeparam>
        /// <param name="field">Reference to the backing field.</param>
        /// <param name="value">The new value.</param>
        /// <param name="name">The name of the property.</param>
        /// <returns>True if the value was changed; otherwise, false.</returns>
        protected bool SetProperty<T>(ref T field, T value, [CallerMemberName] string name = null!)
        {
            if (Equals(field, value)) return false;
            field = value;
            OnPropertyChanged(name);
            return true;
        }
        #endregion
    }
}
```

Listing 33: SettingsTabViewModel.cs (MainViewModel) missing

```
  using   LiveChartsCore;
using LiveChartsCore.SkiaSharpView;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Windows.Input;
using TransDemo.Logic.Services;
using TransDemo.Models;
using TransDemo.UI.ViewModels;

using LiveChartsCore.SkiaSharpView.Painting;
using SkiaSharp;


/// <summary>
/// ViewModel for the dashboard, providing chart data and options for the dashboard view.
/// Handles loading and formatting of statistics for various chart types.
/// </summary>
public class DashboardViewModel : INotifyPropertyChanged
{
    private readonly DashboardStatsService _statsService;

    /// <summary>
    /// Enum representing the available chart types for the dashboard.
    /// </summary>
    public enum DashboardChartType
    {
        /// <summary>
        /// Daily transaction count chart.
        /// </summary>
        DailyTxnCount,
        /// <summary>
        /// Daily transaction amount chart.
        /// </summary>
        DailyTxnAmount,
        /// <summary>
        /// Daily cumulative balance chart.
        /// </summary>
        DailyCumulativeBalance,
        /// <summary>
        /// Pie chart of branch transaction shares.
        /// </summary>
```

```
        BranchPie,
        /// <summary>
        /// Column chart of top customers by transaction amount.
        /// </summary>
        TopCustomers
    }

    /// <summary>
    /// Gets the collection of chart series to be displayed.
    /// </summary>
    public ObservableCollection<ISeries> ChartSeries { get; } = new();

    /// <summary>
    /// Gets the X axes for the chart.
    /// </summary>
    public Axis[] XAxes { get; } =
    {
        new Axis { Labels = [] }
    };

    /// <summary>
    /// Gets the Y axes for the chart.
    /// </summary>
    public ObservableCollection<Axis> YAxes { get; } = new()
    {
        new Axis { Labeler = value => value.ToString("N0") }
    };

    /// <summary>
    /// Gets the available day options for the dashboard statistics.
    /// </summary>
    public ObservableCollection<int> DayOptions { get; } = new() { 7, 14, 30, 60 };

    /// <summary>
    /// Gets the available chart types for the dashboard.
    /// </summary>
    public ObservableCollection<DashboardChartType> ChartTypes { get; } = new()
    {
        DashboardChartType.DailyTxnCount,
        DashboardChartType.DailyTxnAmount,
        DashboardChartType.DailyCumulativeBalance,
        DashboardChartType.BranchPie,
        DashboardChartType.TopCustomers
    };

    private int _selectedDays = 7;
    /// <summary>
    /// Gets or sets the number of days to display in the chart.
    /// </summary>
    public int SelectedDays
    {
        get => _selectedDays;
        set { if (SetProperty(ref _selectedDays, value)) _ = LoadChartAsync(); }
    }

    private DashboardChartType _selectedChartType = DashboardChartType.DailyTxnCount;
    /// <summary>
    /// Gets or sets the selected chart type.
    /// </summary>
    public DashboardChartType SelectedChartType
    {
        get => _selectedChartType;
        set { if (SetProperty(ref _selectedChartType, value)) _ = LoadChartAsync(); }
    }

    /// <summary>
    /// Gets the command to refresh the chart data.
    /// </summary>
    public ICommand RefreshCommand { get; }

    /// <summary>
    /// Initializes a new instance of the <see cref="DashboardViewModel"/> class.
    /// </summary>
    /// <param name="statsService">The service used to retrieve dashboard statistics.</param>
    public DashboardViewModel(DashboardStatsService statsService)
    {
        _statsService = statsService;
        RefreshCommand = new RelayCommand(async _ => await LoadChartAsync());
        _ = LoadChartAsync();
```

```
122     }
123
124     /// <summary>
125     /// Loads and prepares chart data asynchronously based on the selected chart type and days.
126     /// </summary>
127     private async Task LoadChartAsync()
128     {
129         var rawStats = (await _statsService.GetStatsAsync(SelectedDays)).ToList();
130
131         // Create a list of all dates from today to -N days
132         var allDates = Enumerable.Range(0, SelectedDays)
133             .Select(offset => DateTime.Today.AddDays(-offset))
134             .OrderBy(d => d)
135             .ToList();
136
137         // Dictionary of statistics by date
138         var statsDict = rawStats.ToDictionary(s => s.TxnDate.Date);
139
140         // Fill missing days with zero values
141         var filledStats = allDates.Select(date =>
142             statsDict.TryGetValue(date, out var stat)
143                 ? new DailyTransactionStat { TxnDate = date, TxnCount = stat.TxnCount, TotalAmount = stat
    .TotalAmount }
144                 : new DailyTransactionStat { TxnDate = date, TxnCount = 0, TotalAmount = 0 }
145         ).ToList();
146
147         // Update X axis labels
148         XAxes[0].Labels = filledStats.Select(s => s.TxnDate.ToString("MM-dd")).ToArray();
149         OnPropertyChanged(nameof(XAxes));
150
151         // Clear previous series
152         ChartSeries.Clear();
153
154         // Build chart series based on selected chart type
155         switch (SelectedChartType)
156         {
157             case DashboardChartType.DailyTxnCount:
158                 ChartSeries.Add(new LineSeries<int>
159                 {
160                     Values = filledStats.Select(s => s.TxnCount).ToArray(),
161                     Name = "Liczba transakcji dziennie"
162                 });
163                 break;
164
165             case DashboardChartType.DailyTxnAmount:
166                 ChartSeries.Add(new LineSeries<decimal>
167                 {
168                     Values = filledStats.Select(s => s.TotalAmount).ToArray(),
169                     Name = "Suma transakcji dziennie"
170                 });
171                 break;
172
173             case DashboardChartType.DailyCumulativeBalance:
174                 {
175                     var rawBalances = (await _statsService.GetDailyBalanceAsync(SelectedDays)).ToList();
176                     ChartSeries.Add(new LineSeries<decimal>
177                     {
178                         Values = rawBalances.OrderBy(b => b.BalanceDate).Select(b => b.CumulativeBalance)
    .ToArray(),
179                         Name = "Saldo sumaryczne dziennie"
180                     });
181
182                     XAxes[0].Labels = rawBalances.OrderBy(b => b.BalanceDate).Select(b => b.BalanceDate.
    ToString("MM-dd")).ToArray();
183
184                     break;
185                 }
186
187             case DashboardChartType.BranchPie:
188                 {
189                     var shares = (await _statsService.GetBranchShareAsync()).ToList();
190
191                     foreach (var s in shares)
192                     {
193                         ChartSeries.Add(new PieSeries<decimal>
194                         {
195                             Values = [s.SharePercent],
196                             Name = s.BranchName,
197                             DataLabelsSize = 14,
```

```
                            DataLabelsPosition = LiveChartsCore.Measure.PolarLabelsPosition.Middle,
                            DataLabelsFormatter = chartPoint => $"{chartPoint.Model:F2} %",
                            DataLabelsPaint = new SolidColorPaint(SKColors.Black),
                        });
                    }

                    // Pie charts don't need axes
                    XAxes[0].Labels = [];
                    YAxes.Clear();
                    YAxes.Add(new Axis { Labeler = value => value.ToString("N0") });
                    break;
                }

            case DashboardChartType.TopCustomers:
                {
                    var top = (await _statsService.GetTopCustomersAsync()).ToList();
                    ChartSeries.Add(new ColumnSeries<decimal>
                    {
                        Values = top.Select(c => c.TotalAmount).ToArray(),
                        Name = "Top klienci"
                    });

                    XAxes[0].Labels = top.Select(c => c.FullName).ToArray();
                    break;
                }
        }
    }

    /// <summary>
    /// Occurs when a property value changes.
    /// </summary>
    public event PropertyChangedEventHandler? PropertyChanged;

    /// <summary>
    /// Sets the property and raises the PropertyChanged event if the value changes.
    /// </summary>
    /// <typeparam name="T">The type of the property.</typeparam>
    /// <param name="field">The backing field reference.</param>
    /// <param name="value">The new value.</param>
    /// <param name="name">The property name.</param>
    /// <returns>True if the value was changed; otherwise, false.</returns>
    protected bool SetProperty<T>(ref T field, T value, [CallerMemberName] string? name = null)
    {
        if (Equals(field, value)) return false;
        field = value;
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
        return true;
    }

    /// <summary>
    /// Raises the PropertyChanged event for the specified property.
    /// </summary>
    /// <param name="name">The property name.</param>
    protected void OnPropertyChanged(string name) => PropertyChanged?.Invoke(this, new
    PropertyChangedEventArgs(name));
}
```

Listing 34: DashboardViewModel.cs

```
 using   System;
using System.Collections.ObjectModel;
using System.Linq;
using System.Windows.Input;
using TransDemo.Data.Repositories;
using TransDemo.Logic.Services;
using TransDemo.Models;
using TransDemo.UI.Views;
using Microsoft.Extensions.DependencyInjection;
using System.Windows;


namespace TransDemo.UI.ViewModels
{
    /// <summary>
    /// ViewModel responsible for handling transfer operations between accounts.
    /// Provides properties and commands for selecting accounts, entering transfer details, and executing
     transfers.
    /// </summary>
    public class TransfersViewModel : BaseViewModel
    {
```

```
        private readonly IAccountRepository _acctRepo;
        private readonly ICustomerRepository _custRepo;
        private readonly TransferService _transferSvc;

        /// <summary>
        /// Gets the collection of all available accounts.
        /// </summary>
        public ObservableCollection<Account> Accounts { get; } = new();

        private Account? _from;
        /// <summary>
        /// Gets or sets the selected source account for the transfer.
        /// </summary>
        public Account? SelectedFromAccount
        {
            get => _from;
            set
            {
                if (SetProperty(ref _from, value))
                    OnPropertyChanged(nameof(CanExecuteTransfer));
            }
        }

        private Account? _to;
        /// <summary>
        /// Gets or sets the selected destination account for the transfer.
        /// </summary>
        public Account? SelectedToAccount
        {
            get => _to;
            set
            {
                SetProperty(ref _to, value);
                OnPropertyChanged(nameof(CanExecuteTransfer));
            }
        }

        private decimal _amount;
        /// <summary>
        /// Gets or sets the transfer amount as a string. Validates and updates the internal decimal
    value.
        /// </summary>
        public string TransferAmount
        {
            get => _amount.ToString();
            set
            {
                if (decimal.TryParse(value, out var d))
                {
                    _amount = d;
                    ErrorMessage = "";
                }
                else
                {
                    ErrorMessage = "Nieprawidłowa kwota.";
                }
                OnPropertyChanged(nameof(TransferAmount));
                OnPropertyChanged(nameof(CanExecuteTransfer));
            }
        }

        private string _description = "";
        /// <summary>
        /// Gets or sets the description for the transfer.
        /// </summary>
        public string TransferDescription
        {
            get => _description;
            set => SetProperty(ref _description, value);
        }

        private string _error = "";
        /// <summary>
        /// Gets or sets the error message to display in the UI.
        /// </summary>
        public string ErrorMessage
        {
            get => _error;
            set => SetProperty(ref _error, value);
```

```
99          }
100
101         /// <summary>
102         /// Gets a value indicating whether the transfer can be executed based on current selections and
    amount.
103         /// </summary>
104         public bool CanExecuteTransfer =>
105                SelectedFromAccount != null
106             && SelectedToAccount != null
107             && SelectedFromAccount.AccountId != SelectedToAccount.AccountId
108             && _amount > 0
109             && SelectedFromAccount.Balance >= _amount;
110
111         /// <summary>
112         /// Command to search and select the source account.
113         /// </summary>
114         public ICommand SearchFromAccountCommand { get; }
115         /// <summary>
116         /// Command to search and select the destination account.
117         /// </summary>
118         public ICommand SearchToAccountCommand { get; }
119         /// <summary>
120         /// Command to execute the transfer.
121         /// </summary>
122         public ICommand ExecuteTransferCommand { get; }
123         /// <summary>
124         /// Command to execute the transfer with simulated error.
125         /// </summary>
126         public ICommand ExecuteTransferWithErrorCommand { get; }
127
128         /// <summary>
129         /// Initializes a new instance of the <see cref="TransfersViewModel"/> class.
130         /// </summary>
131         /// <param name="acctRepo">The account repository.</param>
132         /// <param name="custRepo">The customer repository.</param>
133         /// <param name="transferSvc">The transfer service.</param>
134         public TransfersViewModel(
135             IAccountRepository acctRepo,
136             ICustomerRepository custRepo,
137             TransferService transferSvc)
138         {
139             _acctRepo = acctRepo;
140             _custRepo = custRepo;
141             _transferSvc = transferSvc;
142
143             SearchFromAccountCommand = new RelayCommand(_ => PickClientAndAccount(a =>
    SelectedFromAccount = a));
144             SearchToAccountCommand = new RelayCommand(_ => PickClientAndAccount(a => SelectedToAccount =
    a));
145             ExecuteTransferCommand = new RelayCommand(_ => Execute(false), _ => CanExecuteTransfer);
146             ExecuteTransferWithErrorCommand = new RelayCommand(_ => Execute(true), _ =>
    CanExecuteTransfer);
147
148             LoadAccounts();
149         }
150
151         /// <summary>
152         /// Loads all accounts from the repository into the <see cref="Accounts"/> collection.
153         /// </summary>
154         private void LoadAccounts()
155         {
156             Accounts.Clear();
157             foreach (var a in _acctRepo.GetAll())
158                 Accounts.Add(a);
159         }
160
161         /// <summary>
162         /// Opens a dialog to select a customer and assigns the corresponding account using the provided
    action.
163         /// </summary>
164         /// <param name="assign">The action to assign the selected account.</param>
165         private void PickClientAndAccount(Action<Account> assign)
166         {
167             // Pobieramy VM i widok z DI
168             var vm = App.Services.GetRequiredService<SearchCustomerViewModel>();
169             var dlg = App.Services.GetRequiredService<SearchCustomerView>();
170
171             // Podłączamy ViewModel
172             dlg.DataContext = vm;
```

```
            // Ustawiamy właściciela (MainWindow) i dzięki CenterOwner OKNO będzie wyśrodkowane
            dlg.Owner = Application.Current.MainWindow;

            // Pokaż dialog modalnie
            if (dlg.ShowDialog() == true && vm.SelectedCustomer != null)
            {
                var acc = Accounts.FirstOrDefault(x => x.CustomerId == vm.SelectedCustomer.CustomerId);
                if (acc != null) assign(acc);
            }
        }

        /// <summary>
        /// Executes the transfer operation using the transfer service.
        /// Optionally simulates an error if <paramref name="simulateError"/> is true.
        /// Updates the error message and reloads accounts after execution.
        /// </summary>
        /// <param name="simulateError">If true, simulates an error during transfer.</param>
        private void Execute(bool simulateError)
        {
            ErrorMessage = "";
            try
            {
                _transferSvc.ExecuteDistributedTransfer(
                    SelectedFromAccount!, SelectedToAccount!, _amount, simulateError);

                // odświeżamy salda
                LoadAccounts();
            }
            catch (Exception ex)
            {
                ErrorMessage = ex.Message;
            }
        }
    }
}
```

Listing 35: TransfersViewModel.cs

```
 using   System.Collections.ObjectModel;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Threading.Tasks;
using TransDemo.Logic.Services;
using TransDemo.Models;

namespace TransDemo.UI.ViewModels
{
    /// <summary>
    /// ViewModel responsible for managing and exposing transaction history data for a selected branch.
    /// </summary>
    public class HistoryViewModel : INotifyPropertyChanged
    {
        /// <summary>
        /// Service used to query transaction history from the database.
        /// </summary>
        private readonly HistoryQueryService _historyService;

        /// <summary>
        /// Collection of history entries for the currently selected branch.
        /// </summary>
        public ObservableCollection<HistoryEntry> Entries { get; } = new();

        private int _selectedBranch = 0;

        /// <summary>
        /// Gets or sets the currently selected branch number.
        /// When changed, triggers reloading of the history entries.
        /// </summary>
        public int SelectedBranch
        {
            get => _selectedBranch;
            set
            {
                if (SetProperty(ref _selectedBranch, value))
                    _ = LoadHistoryAsync();
            }
        }

        /// <summary>
```

```
        /// Initializes a new instance of the <see cref="HistoryViewModel"/> class.
        /// Loads the history for the default branch on creation.
        /// </summary>
        /// <param name="historyService">Service for querying history data.</param>
        public HistoryViewModel(HistoryQueryService historyService)
        {
            _historyService = historyService;
            _ = LoadHistoryAsync();
        }

        /// <summary>
        /// Asynchronously loads the transaction history for the currently selected branch.
        /// Clears the existing entries and populates the collection with new data.
        /// </summary>
        /// <returns>A task representing the asynchronous operation.</returns>
        public async Task LoadHistoryAsync()
        {
            Entries.Clear();
            var entries = await _historyService.GetBranchHistoryAsync(SelectedBranch);
            foreach (var entry in entries)
                Entries.Add(entry);
        }

        /// <summary>
        /// Event raised when a property value changes.
        /// </summary>
        public event PropertyChangedEventHandler? PropertyChanged;

        /// <summary>
        /// Sets the property to the specified value and raises the <see cref="PropertyChanged"/> event
    if the value has changed.
        /// </summary>
        /// <typeparam name="T">Type of the property.</typeparam>
        /// <param name="field">Reference to the backing field.</param>
        /// <param name="value">New value to set.</param>
        /// <param name="name">Name of the property (automatically supplied by the compiler).</param>
        /// <returns>True if the value was changed; otherwise, false.</returns>
        protected bool SetProperty<T>(ref T field, T value, [CallerMemberName] string? name = null)
        {
            if (Equals(field, value)) return false;
            field = value;
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
            return true;
        }
    }
}
```

Listing 36: HistoryViewModel.cs

```
  using   System;
using System.Collections.ObjectModel;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Input;
using Microsoft.Data.SqlClient;
using TransDemo.UI.Models;
using System.Data;

namespace TransDemo.UI.ViewModels
{
    /// <summary>
    /// ViewModel responsible for managing application database connection settings.
    /// Handles loading, saving, testing, and editing connection configurations.
    /// </summary>
    public class SettingsViewModel : BaseViewModel
    {
        /// <summary>
        /// Path to the application settings file.
        /// </summary>
        private readonly string _settingsPath = "appsettings.json";

        private ObservableCollection<DbConnectionSetting> _connections = [];

        /// <summary>
        /// Gets or sets the collection of database connection settings.
        /// </summary>
        public ObservableCollection<DbConnectionSetting> Connections
        {
            get => _connections;
```

```
            set => SetProperty(ref _connections, value);
        }

        /// <summary>
        /// Gets the list of available databases for the selected connection.
        /// </summary>
        public ObservableCollection<string> AvailableDatabases { get; } = [];

        /// <summary>
        /// Command to test the selected database connection.
        /// </summary>
        public ICommand TestConnectionCommand { get; }

        /// <summary>
        /// Command to apply and save the current connection settings.
        /// </summary>
        public ICommand ApplyCommand { get; }

        /// <summary>
        /// Command to cancel and close the settings window.
        /// </summary>
        public ICommand CancelCommand { get; }

        private DbConnectionSetting _selectedConnection;

        /// <summary>
        /// Gets or sets the currently selected database connection setting.
        /// When changed, reloads the list of available databases for the new connection.
        /// </summary>
        public DbConnectionSetting? SelectedConnection
        {
            get => _selectedConnection;
            set
            {
                if (SetProperty(ref _selectedConnection, value) && value != null)
                {
                    // Reload the list of databases whenever the connection changes
                    LoadDatabasesForConnection(value);
                }
            }
        }

        /// <summary>
        /// Command to add a new database connection setting.
        /// </summary>
        public ICommand AddConnectionCommand { get; }

        /// <summary>
        /// Command to remove the selected database connection setting.
        /// </summary>
        public ICommand RemoveConnectionCommand { get; }

        /// <summary>
        /// Initializes a new instance of the <see cref="SettingsViewModel"/> class.
        /// Loads connection settings and sets up commands.
        /// </summary>
        public SettingsViewModel()
        {
            // 1) Load the dictionary of connection strings from settings
            var raw = AppSettings.Load(_settingsPath).ConnectionStrings;

            // 2) Parse each entry into a DbConnectionSetting
            var list = raw.Select(kvp =>
            {
                var sb = new SqlConnectionStringBuilder(kvp.Value);
                return new DbConnectionSetting
                {
                    Key = kvp.Key,
                    Server = sb.DataSource,
                    Database = sb.InitialCatalog,
                    UserName = sb.UserID,
                    Password = sb.Password,
                    Encrypt = sb.Encrypt,
                    TrustServerCertificate = sb.TrustServerCertificate,
                    RememberPassword = !string.IsNullOrEmpty(sb.Password),
                    KeepConnectionAlive = false
                };
            }).ToList();

```

```
            Connections = [.. list];
            // 4) Set the default SelectedConnection and immediately load its databases
            SelectedConnection = Connections.FirstOrDefault();

            // 3) Initialize commands
            TestConnectionCommand = new RelayCommand<DbConnectionSetting>(async c =>
                await TestConnectionAsync(c)
            );

            ApplyCommand = new RelayCommand<object>(_ =>
            {
                // 4) Flatten back to Dictionary<string,string>
                var dict = Connections.ToDictionary(
                    c => c.Key,
                    c =>
                    {
                        var sb = new SqlConnectionStringBuilder
                        {
                            DataSource = c.Server,
                            InitialCatalog = c.Database,
                            UserID = c.UserName,
                            Password = c.RememberPassword ? c.Password : "",
                            Encrypt = c.Encrypt,
                            TrustServerCertificate = c.TrustServerCertificate,
                            ConnectTimeout = 30
                        };
                        return sb.ConnectionString;
                    });

                var updated = new AppSettings { ConnectionStrings = dict };
                updated.Save(_settingsPath);

                MessageBox.Show("Ustawienia zapisane.",
                                "OK",
                                MessageBoxButton.OK,
                                MessageBoxImage.Information);
            });

            CancelCommand = new RelayCommand<Window>(w => w?.Close());

            AddConnectionCommand = new RelayCommand<object>(_ =>
            {
                // Add a new, empty entry
                var newConn = new DbConnectionSetting
                {
                    Key = "NewConnection",
                    Server = "",
                    Database = "",
                    UserName = "",
                    Password = "",
                    RememberPassword = false,
                    KeepConnectionAlive = false,
                    Encrypt = false,
                    TrustServerCertificate = false
                };
                Connections.Add(newConn);
            });

            RemoveConnectionCommand = new RelayCommand<DbConnectionSetting>(conn =>
            {
                if (conn != null)
                    Connections.Remove(conn);
            }, conn => conn != null);
        }

        /// <summary>
        /// Tests the database connection for the specified connection setting.
        /// Shows a message box with the result.
        /// </summary>
        /// <param name="c">The connection setting to test.</param>
        private async Task TestConnectionAsync(DbConnectionSetting c)
        {
            var sb = new SqlConnectionStringBuilder
            {
                DataSource = c.Server,
                InitialCatalog = c.Database,
                UserID = c.UserName,
                Password = c.Password,
                Encrypt = c.Encrypt,
```

```
                TrustServerCertificate = c.TrustServerCertificate,
                ConnectTimeout = 5
            };

            using var conn = new SqlConnection(sb.ConnectionString);
            try
            {
                await conn.OpenAsync();
                MessageBox.Show($"Połączono z    {c.Key}   .",
                                "Test OK",
                                MessageBoxButton.OK,
                                MessageBoxImage.Information);
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message,
                                "Błąd połączenia",
                                MessageBoxButton.OK,
                                MessageBoxImage.Error);
            }
        }

        /// <summary>
        /// Loads the list of database names from the server specified in the given connection setting.
        /// Updates the <see cref="AvailableDatabases"/> collection.
        /// </summary>
        /// <param name="c">The connection setting for which to load databases.</param>
        private void LoadDatabasesForConnection(DbConnectionSetting c)
        {
            AvailableDatabases.Clear();
            // Build connection string to master database
            var sb = new SqlConnectionStringBuilder
            {
                DataSource = c.Server,
                InitialCatalog = "master",
                UserID = c.UserName,
                Password = c.Password,
                Encrypt = c.Encrypt,
                TrustServerCertificate = c.TrustServerCertificate,
                ConnectTimeout = 5
            };
            using var conn = new SqlConnection(sb.ConnectionString);
            conn.Open();
            using var cmd = conn.CreateCommand();
            cmd.CommandText = "SELECT name FROM sys.databases ORDER BY name";
            using var rdr = cmd.ExecuteReader();
            while (rdr.Read())
                AvailableDatabases.Add(rdr.GetString(0));

            // Set the default database to the one in configuration, or the first available
            if (AvailableDatabases.Contains(c.Database))
                c.Database = c.Database;
            else if (AvailableDatabases.Count > 0)
                c.Database = AvailableDatabases[0];
        }
    }
}
```

Listing 37: SettingsViewModel.cs

```
 using   System;
using System.Collections.ObjectModel;
using System.Linq;
using System.Windows;
using System.Windows.Input;
using TransDemo.Data.Repositories;
using TransDemo.Models;

namespace TransDemo.UI.ViewModels
{
    /// <summary>
    /// ViewModel responsible for searching and selecting customers.
    /// Provides filtering, selection, and dialog result logic for customer search dialogs.
    /// </summary>
    public class SearchCustomerViewModel : BaseViewModel
    {
        /// <summary>
        /// Repository for accessing customer data.
        /// </summary>
        private readonly ICustomerRepository _custRepo;
```

```
21
22          /// <summary>
23          /// Gets the collection of customers matching the current filter.
24          /// </summary>
25          public ObservableCollection<Customer> Customers { get; } = new();
26
27          /// <summary>
28          /// Command to confirm the selection of a customer.
29          /// </summary>
30          public ICommand ConfirmCommand { get; }
31
32          /// <summary>
33          /// Command to cancel the customer selection dialog.
34          /// </summary>
35          public ICommand CancelCommand { get; }
36
37          private string _filterText = "";
38
39          /// <summary>
40          /// Gets or sets the filter text used to search for customers.
41          /// Setting this property reloads the customer list.
42          /// </summary>
43          public string FilterText
44          {
45              get => _filterText;
46              set
47              {
48                  SetProperty(ref _filterText, value);
49                  LoadCustomers();
50              }
51          }
52
53          private Customer? _selected;
54
55          /// <summary>
56          /// Gets or sets the currently selected customer.
57          /// </summary>
58          public Customer? SelectedCustomer
59          {
60              get => _selected;
61              set => SetProperty(ref _selected, value);
62          }
63
64          /// <summary>
65          /// Initializes a new instance of the <see cref="SearchCustomerViewModel"/> class.
66          /// </summary>
67          /// <param name="custRepo">The customer repository to use for data access.</param>
68          public SearchCustomerViewModel(ICustomerRepository custRepo)
69          {
70              _custRepo = custRepo;
71              ConfirmCommand = new RelayCommand(_ => Confirm(), _ => SelectedCustomer != null);
72              CancelCommand = new RelayCommand(_ => Cancel());
73              LoadCustomers();
74          }
75
76          /// <summary>
77          /// Loads customers from the repository using the current filter text.
78          /// Clears and repopulates the <see cref="Customers"/> collection.
79          /// </summary>
80          private void LoadCustomers()
81          {
82              Customers.Clear();
83              foreach (var c in _custRepo.GetAll(FilterText))
84                  Customers.Add(c);
85          }
86
87          /// <summary>
88          /// Confirms the selection and closes the dialog with a positive result.
89          /// </summary>
90          private void Confirm()
91          {
92              CloseDialog(true);
93          }
94
95          /// <summary>
96          /// Cancels the selection and closes the dialog with a negative result.
97          /// </summary>
98          private void Cancel()
99          {
```

```
            CloseDialog(false);
        }

        /// <summary>
        /// Closes the dialog window associated with this ViewModel.
        /// Sets the dialog result to the specified value.
        /// </summary>
        /// <param name="result">The dialog result to set (true for confirm, false for cancel).</param>
        private void CloseDialog(bool result)
        {
            if (Application.Current.Windows
                    .OfType<Window>()
                    .FirstOrDefault(w => w.DataContext == this)
                is Window wnd)
            {
                wnd.DialogResult = result;
                wnd.Close();
            }
        }
    }
}
```

Listing 38: SearchCustomerViewModel.cs

TransDemo.UI.Views (code-behind plików XAML):

```
   // MainWindow.xaml.cs
using System;
using System.Windows;
using System.Windows.Media;
using Microsoft.Extensions.Configuration;
using MaterialDesignThemes.Wpf;
using TransDemo.Data.Repositories;
using TransDemo.Logic.Services;
using TransDemo.UI.ViewModels;


namespace TransDemo.UI
{
    /// <summary>
    /// Interaction logic for MainWindow.
    /// This is the main window of the application, responsible for initializing the UI and setting up
    the theme.
    /// </summary>
    public partial class MainWindow : Window
    {
        /// <summary>
        /// Initializes a new instance of the <see cref="MainWindow"/> class.
        /// Sets the DataContext to the provided <see cref="MainViewModel"/> and configures the Material
    Design theme.
        /// </summary>
        /// <param name="vm">The main view model for the application.</param>
        public MainWindow(MainViewModel vm)
        {
            InitializeComponent();
            DataContext = vm;

            // Theme configuration using MaterialDesignThemes.Wpf
            // Sets the base theme to Light, primary color to Indigo, and secondary color to Lime.
            var palette = new PaletteHelper();
            var theme = palette.GetTheme();
            theme.SetBaseTheme(BaseTheme.Light);
            theme.SetPrimaryColor(Colors.Indigo);
            theme.SetSecondaryColor(Colors.Lime);
            palette.SetTheme(theme);
        }
    }
}
```

Listing 39: MainWindow.xaml.cs

```
   // DashboardView.xaml.cs
using System.Windows.Controls;

namespace TransDemo.UI.Views
{
    /// <summary>
    /// Interaction logic for <see cref="DashboardView"/>.
    /// This view is responsible for displaying the dashboard UI in the application.
    /// The DataContext is injected via IoC/Prism and is expected to be an instance of
    DashboardTabViewModel.
```

```
    /// </summary>
    public partial class DashboardView : UserControl
    {
        /// <summary>
        /// Initializes a new instance of the <see cref="DashboardView"/> class.
        /// Sets up the component and prepares it for use in the UI.
        /// </summary>
        public DashboardView()
        {
            InitializeComponent();
            // DataContext is injected from DashboardTabViewModel via IoC/Prism
        }
    }
}
```

Listing 40: DashboardView.xaml.cs

```
  using   System.Windows.Controls;

namespace TransDemo.UI.Views
{
    /// <summary>
    /// Interaction logic for TransfersView.xaml.
    /// This class represents the view for displaying and managing transfers in the application.
    /// </summary>
    public partial class TransfersView : UserControl
    {
        /// <summary>
        /// Initializes a new instance of the <see cref="TransfersView"/> class.
        /// Sets up the UI components defined in the corresponding XAML file.
        /// </summary>
        public TransfersView()
        {
            InitializeComponent();
        }
    }
}
```

Listing 41: TransfersView.xaml.cs

```
  using   System.Windows.Controls;

namespace TransDemo.UI.Views
{
    /// <summary>
    /// Interaction logic for HistoryView.xaml.
    /// This class represents the view for displaying history in the application.
    /// </summary>
    public partial class HistoryView : UserControl
    {
        /// <summary>
        /// Initializes a new instance of the <see cref="HistoryView"/> class.
        /// </summary>
        public HistoryView()
        {
            InitializeComponent();
        }

        /// <summary>
        /// Handles the SelectionChanged event of the DataGrid control.
        /// This method is called whenever the selection changes in the associated DataGrid.
        /// </summary>
        /// <param name="sender">The source of the event, typically the DataGrid.</param>
        /// <param name="e">The event data containing information about the selection change.</param>
        private void DataGrid_SelectionChanged(object sender, SelectionChangedEventArgs e)
        {

        }
    }
}
```

Listing 42: HistoryView.xaml.cs

```
  using   System.Windows;
using TransDemo.UI.ViewModels;

namespace TransDemo.UI.Views
{
    /// <summary>
```

```
    /// Interaction logic for <see cref="SettingsView"/>.
    /// This window allows the user to view and modify application database connection settings.
    /// </summary>
    public partial class SettingsView : Window
    {
        /// <summary>
        /// Initializes a new instance of the <see cref="SettingsView"/> class.
        /// Sets the data context to the provided <see cref="SettingsViewModel"/>.
        /// </summary>
        /// <param name="vm">The view model containing logic and data for the settings view.</param>
        public SettingsView(SettingsViewModel vm)
        {
            InitializeComponent();
            DataContext = vm;
        }
    }
}
```

Listing 43: SettingsView.xaml.cs

```
  using   System.Windows;

namespace TransDemo.UI.Views
{
    /// <summary>
    /// Interaction logic for SearchCustomerView.xaml
    /// This window provides a user interface for searching customers within the application.
    /// </summary>
    public partial class SearchCustomerView : Window
    {
        /// <summary>
        /// Initializes a new instance of the <see cref="SearchCustomerView"/> class.
        /// Sets up the UI components and prepares the window for user interaction.
        /// </summary>
        public SearchCustomerView()
        {
            InitializeComponent();
        }
    }
}
```

Listing 44: SearchCustomerView.xaml.cs

E. Skrypty SQL tworzące bazy danych i procedury

CreateCentralDatabase.sql (DDL bazy BankCentral):

```
CREATE TABLE dbo.AccountTypes (
    AccountTypeId INT IDENTITY(1,1) PRIMARY KEY,
    TypeName   NVARCHAR(50) NOT NULL UNIQUE,
    Description NVARCHAR(200) NULL
);
```

Listing 45: Tables/AccountTypes.sql (BankCentral)

```
CREATE TABLE dbo.Branches (
    BranchId    INT IDENTITY(1,1) PRIMARY KEY,
    BranchName  NVARCHAR(100) NOT NULL UNIQUE,
    Address     NVARCHAR(250) NOT NULL,
    CreatedDate DATETIME2 NOT NULL DEFAULT SYSUTCDATETIME()
);
```

Listing 46: Tables/Branches.sql (BankCentral)

```
CREATE TABLE dbo.Customers (
    CustomerId INT IDENTITY(1,1) PRIMARY KEY,
    FirstName  NVARCHAR(100) NOT NULL,
    LastName   NVARCHAR(100) NOT NULL,
    Email      NVARCHAR(256) NOT NULL UNIQUE,
    DateOfBirth DATE NULL,
    CreatedDate DATETIME2 NOT NULL DEFAULT SYSUTCDATETIME()
);
GO

CREATE INDEX IX_Customers_LastName ON dbo.Customers(LastName, FirstName);
```

Listing 47: Tables/Customers.sql (BankCentral)

```sql
CREATE TABLE dbo.Accounts (
    AccountId     INT IDENTITY(1,1) PRIMARY KEY,
    CustomerId    INT NOT NULL,
    AccountNumber VARCHAR(34) NOT NULL UNIQUE,
    Balance       MONEY NOT NULL CONSTRAINT DF_Accounts_Balance DEFAULT(0),
    BranchId      INT NOT NULL,
    AccountTypeId INT NOT NULL,
    CreatedDate   DATETIME2 NOT NULL DEFAULT SYSUTCDATETIME(),
    CONSTRAINT FK_Accounts_Customers    FOREIGN KEY(CustomerId)    REFERENCES dbo.Customers(CustomerId),
    CONSTRAINT FK_Accounts_Branches     FOREIGN KEY(BranchId)      REFERENCES dbo.Branches(BranchId),
    CONSTRAINT FK_Accounts_AccountTypes FOREIGN KEY(AccountTypeId) REFERENCES dbo.AccountTypes(
    AccountTypeId)
);

GO

CREATE INDEX IX_Accounts_CustomerId ON dbo.Accounts(CustomerId);
```

Listing 48: Tables/Accounts.sql (BankCentral)

```sql
CREATE TABLE dbo.Transactions (
    TransactionId    UNIQUEIDENTIFIER DEFAULT NEWID() PRIMARY KEY,
    AccountId        INT NOT NULL,
    BranchId         INT NOT NULL,
    Amount           MONEY NOT NULL,
    TransactionType  CHAR(10) NOT NULL CHECK (TransactionType IN ('DEBIT','CREDIT')),
    CreatedAt        DATETIME2 NOT NULL DEFAULT SYSUTCDATETIME(),
    Description      NVARCHAR(200) NULL,
    CONSTRAINT FK_Transactions_Accounts FOREIGN KEY(AccountId)  REFERENCES dbo.Accounts(AccountId),
    CONSTRAINT FK_Transactions_Branches FOREIGN KEY(BranchId)   REFERENCES dbo.Branches(BranchId)
);

GO

CREATE INDEX IX_Transactions_AccountId ON dbo.Transactions(AccountId);
```

Listing 49: Tables/Transactions.sql (BankCentral)

```sql
CREATE TABLE dbo.LoginHistory (
    LoginHistoryId INT IDENTITY(1,1) PRIMARY KEY,
    UserName       NVARCHAR(100) NOT NULL,
    LoginTime      DATETIME2 NOT NULL DEFAULT SYSUTCDATETIME(),
    Success        BIT NOT NULL,
    IPAddress      NVARCHAR(45) NULL
);
```

Listing 50: Tables/LoginHistory.sql (BankCentral)

```sql
CREATE VIEW dbo.vAccountBalances
AS
SELECT
    a.AccountId,
    a.AccountNumber,
    c.FirstName,
    c.LastName,
    b.BranchName,
    at.TypeName AS AccountType,
    a.Balance,
    MAX(t.CreatedAt) AS LastTxnDate
FROM dbo.Accounts a
JOIN dbo.Customers c ON a.CustomerId = c.CustomerId
JOIN dbo.Branches b ON a.BranchId = b.BranchId
JOIN dbo.AccountTypes at ON a.AccountTypeId = at.AccountTypeId
LEFT JOIN dbo.Transactions t ON a.AccountId = t.AccountId
GROUP BY
    a.AccountId,
    a.AccountNumber,
    c.FirstName,
    c.LastName,
    b.BranchName,
    at.TypeName,
    a.Balance;
```

Listing 51: Views/vAccountBalances.sql

```sql
CREATE VIEW dbo.vDailyBalances
AS
WITH DailyNet AS (
    SELECT
```

```
        CAST(CreatedAt AS DATE) AS BalanceDate,
        SUM(CASE WHEN TransactionType = 'CREDIT' THEN Amount ELSE -Amount END) AS NetAmount
    FROM dbo.Transactions
    GROUP BY CAST(CreatedAt AS DATE)
)
SELECT
    BalanceDate,
    SUM(NetAmount) OVER (ORDER BY BalanceDate ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
    CumulativeBalance
FROM DailyNet;
```

Listing 52: Views/vDailyBalances.sql

```
CREATE VIEW dbo.vDailyTransactionStats
AS
SELECT
    CAST(CreatedAt AS DATE) AS TxnDate,
    COUNT(*) AS TxnCount,
    SUM(Amount) AS TotalAmount
FROM dbo.Transactions
GROUP BY CAST(CreatedAt AS DATE);
```

Listing 53: Views/vDailyTransactionStats.sql

```
CREATE VIEW dbo.vBranchTransactionShares
AS
WITH Total AS (
    SELECT SUM(Amount) AS TotalAmount FROM dbo.Transactions
)
SELECT
    b.BranchName,
    COUNT(t.TransactionId) AS TxnCount,
    SUM(t.Amount) AS BranchAmount,
    CAST(100.0 * SUM(t.Amount) / tAll.TotalAmount AS DECIMAL(5,2)) AS SharePercent
FROM dbo.Transactions t
JOIN dbo.Branches b ON t.BranchId = b.BranchId
CROSS JOIN Total tAll
GROUP BY b.BranchName, tAll.TotalAmount;
```

Listing 54: Views/vBranchTransactionsShares.sql

```
CREATE VIEW dbo.vTopCustomerTransactions
AS
SELECT
    c.FirstName + ' ' + c.LastName AS FullName,
    SUM(t.Amount) AS TotalAmount
FROM dbo.Customers c
JOIN dbo.Accounts a ON c.CustomerId = a.CustomerId
JOIN dbo.Transactions t ON a.AccountId = t.AccountId
GROUP BY c.FirstName, c.LastName;
```

Listing 55: Views/vTopCustomerTransactions.sql

```
CREATE PROCEDURE dbo.sp_TransferCentral
    @FromAccId INT,
    @ToAccId   INT,
    @Amount    MONEY
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRANSACTION;

        DECLARE @FromBalance MONEY;
        SELECT @FromBalance = Balance FROM dbo.Accounts WHERE AccountId = @FromAccId;

        IF @FromBalance < @Amount
        BEGIN

            ROLLBACK TRANSACTION;
            RAISERROR(N'Niewystarczaj?ce ?rodki na ko?cie ?r?d?owym.', 16, 1);
            RETURN;
        END

        UPDATE dbo.Accounts
           SET Balance = Balance - @Amount
         WHERE AccountId = @FromAccId;
        UPDATE dbo.Accounts
           SET Balance = Balance + @Amount
```

```
         WHERE AccountId = @ToAccId;

        INSERT INTO dbo.Transactions(AccountId, BranchId, Amount, TransactionType, Description)
        VALUES
            (@FromAccId, (SELECT BranchId FROM dbo.Accounts WHERE AccountId = @FromAccId), @Amount, '
    DEBIT', 'Central transfer'),
            (@ToAccId,   (SELECT BranchId FROM dbo.Accounts WHERE AccountId = @ToAccId),   @Amount, '
    CREDIT','Central transfer');

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;

        DECLARE @ErrMsg NVARCHAR(4000) = ERROR_MESSAGE();
        RAISERROR(@ErrMsg, 16, 1);
    END CATCH
END
```

Listing 56: StoredProcedures/sp_TransferCentral.sql

```
CREATE PROCEDURE dbo.sp_TransferDistributed
    @FromAccId INT,
    @ToAccId   INT,
    @Amount    MONEY
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN DISTRIBUTED TRANSACTION;
    BEGIN TRY
        -- 1) Centralny etap
        EXEC dbo.sp_TransferCentral @FromAccId, @ToAccId, @Amount;

        ------------------------------------------------
        -- 2) Oddzia? 1 (Branch1DB przez link BR1_LINKED)
        ------------------------------------------------
        DECLARE
            @cidFrom INT,
            @sql1    NVARCHAR(MAX),
            @params1 NVARCHAR(100);

        SELECT @cidFrom = CustomerId
          FROM dbo.Accounts
         WHERE AccountId = @FromAccId;

        SET @sql1 = N'
            UPDATE dbo.BranchClients
                SET Balance = Balance - @amt,
                    LastSync = SYSUTCDATETIME()
                WHERE CentralClientId = @cid;

            INSERT INTO dbo.BranchTransactions (BranchClientId, Amount, TxnType, Description)
            SELECT BranchClientId, @amt, N''DEBIT'', N''Transfer out''
              FROM dbo.BranchClients
             WHERE CentralClientId = @cid;

            INSERT INTO dbo.History(Info)
                VALUES(N''Obci??enie konta w wyniku transferu rozproszonego   '' + CAST(@amt AS NVARCHAR)
    );
        ';
        SET @params1 = N'@amt MONEY, @cid INT';

        EXEC [BR1_LINKED].Branch1DB.dbo.sp_executesql
            @stmt   = @sql1,
            @params = @params1,
            @amt    = @Amount,
            @cid    = @cidFrom;

        ------------------------------------------------
        -- 3) Oddzia? 2 (Branch2DB przez link BR2_LINKED)
        ------------------------------------------------
        DECLARE
            @cidTo  INT,
            @sql2   NVARCHAR(MAX),
            @params2 NVARCHAR(100) = N'@amt MONEY, @cid INT';

        SELECT @cidTo = CustomerId
          FROM dbo.Accounts
         WHERE AccountId = @ToAccId;

```

```
        SET @sql2 = N'
            UPDATE dbo.BranchClients
               SET Balance = Balance + @amt,
                   LastSync = SYSUTCDATETIME()
             WHERE CentralClientId = @cid;

            INSERT INTO dbo.BranchTransactions (BranchClientId, Amount, TxnType, Description)
            SELECT BranchClientId, @amt, N''CREDIT'', N''Transfer in''
              FROM dbo.BranchClients
             WHERE CentralClientId = @cid;
            INSERT INTO dbo.History(Info)
                   VALUES(N''Uznanie konta w wyniku transferu rozproszonego   '' + CAST(@amt AS NVARCHAR
    ));
        ';

        EXEC [BR2_LINKED].Branch2DB.dbo.sp_executesql
            @stmt   = @sql2,
            @params = @params2,
            @amt    = @Amount,
            @cid    = @cidTo;

        -- 4) Commit 2PC
        COMMIT TRANSACTION;
        DECLARE
            @fromName NVARCHAR(200),
            @toName NVARCHAR(200),
            @fromAccNo VARCHAR(34),
            @toAccNo   VARCHAR(34);

        -- Dane nadawcy
        SELECT
            @fromName = c.FirstName + ' ' + c.LastName,
            @fromAccNo = a.AccountNumber
        FROM dbo.Accounts a
        JOIN dbo.Customers c ON a.CustomerId = c.CustomerId
        WHERE a.AccountId = @FromAccId;

        -- Dane odbiorcy
        SELECT
            @toName = c.FirstName + ' ' + c.LastName,
            @toAccNo = a.AccountNumber
        FROM dbo.Accounts a
        JOIN dbo.Customers c ON a.CustomerId = c.CustomerId
        WHERE a.AccountId = @ToAccId;

        -- Wpis do historii
        INSERT INTO dbo.History(Info)
        VALUES (
            N'Transfer z ' + @fromName + ' (' + @fromAccNo + ')'
            + N' do ' + @toName + ' (' + @toAccNo + ')'
            + N' - kwota: ' + CAST(@Amount AS NVARCHAR)
        );
    END TRY
    BEGIN CATCH
        IF XACT_STATE() <> 0
            ROLLBACK TRANSACTION;
        THROW;
    END CATCH
END;
```

Listing 57: StoredProcedures/sp_TransferDistributed.sql

```
   /*
Post-Deployment Script Template
--------------------------------------------------------------------------------------
 This file contains SQL statements that will be appended to the build script.
 Use SQLCMD syntax to include a file in the post-deployment script.
 Example:      :r .\myfile.sql
 Use SQLCMD syntax to reference a variable in the post-deployment script.
 Example:      :setvar TableName MyTable
               SELECT * FROM [$(TableName)]
--------------------------------------------------------------------------------------
*/
-- === Typy kont ===
INSERT INTO dbo.AccountTypes (TypeName, Description)
VALUES
('Osobiste', 'Standardowe konto osobiste'),
('Firmowe', 'Konto dla działalności gospodarczej'),
('Oszczędnościowe', 'Konto do oszczędzania z oprocentowaniem');
GO
```

```
19
20 -- === Oddziały ===
21 INSERT INTO dbo.Branches (BranchName, Address)
22 VALUES
23 ('Oddział Centrum', 'ul. Marszałkowska 1, Warszawa'),
24 ('Oddział Wschód', 'ul. Lubelska 42, Lublin');
25 GO
26
27 -- === Klienci ===
28 INSERT INTO dbo.Customers (FirstName, LastName, Email, DateOfBirth)
29 VALUES
30 ('Jan', 'Kowalski', 'jan.k@example.com', '1980-01-15'),
31 ('Anna', 'Nowak', 'anna.n@example.com', '1991-04-12'),
32 ('Piotr', 'Wiśniewski', 'piotr.w@example.com', '1975-09-30'),
33 ('Magda', 'Mazur', 'magda.mazur@example.com', '1988-07-19'),
34 ('Kamil', 'Wójcik', 'kamil.w@example.com', '1993-02-27');
35 GO
36
37 -- === Konta klientów ===
38 INSERT INTO dbo.Accounts (CustomerId, AccountNumber, Balance, BranchId, AccountTypeId)
39 VALUES
40 (1, 'PL00112233445566778899001', 10250.50, 1, 1),
41 (2, 'PL00112233445566778899002', 15300.00, 1, 3),
42 (3, 'PL00112233445566778899003', 74200.00, 2, 2),
43 (4, 'PL00112233445566778899004', 1100.00, 2, 1),
44 (5, 'PL00112233445566778899005', 580.00, 2, 1);
45 GO
46
47 -- === Transakcje kontowe ===
48 INSERT INTO dbo.Transactions (AccountId, BranchId, Amount, TransactionType, Description, CreatedAt)
49 VALUES
50 (1, 1, 250.50, 'CREDIT', 'Wpłata gotówki', DATEADD(DAY, -10, SYSUTCDATETIME())),
51 (1, 1, 500.00, 'DEBIT', 'Płatność za zakupy', DATEADD(DAY, -5, SYSUTCDATETIME())),
52 (2, 1, 15000.00, 'CREDIT', 'Przelew oszczędności', DATEADD(DAY, -20, SYSUTCDATETIME())),
53 (3, 2, 4200.00, 'CREDIT', 'Wpłata z działalności', DATEADD(DAY, -15, SYSUTCDATETIME())),
54 (3, 2, 200.00, 'DEBIT', 'Zakup wyposażenia', DATEADD(DAY, -1, SYSUTCDATETIME())),
55 (5, 2, 100.00, 'DEBIT', 'Opłata za subskrypcję', DATEADD(DAY, -2, SYSUTCDATETIME()));
56 GO
57
58 -- === Historia logowań ===
59 INSERT INTO dbo.LoginHistory (UserName, LoginTime, Success, IPAddress)
60 VALUES
61 ('BranchAppUser',   DATEADD(HOUR, -8, SYSUTCDATETIME()), 1, '192.168.1.101'),
62 ('BranchAppUser',   DATEADD(HOUR, -3, SYSUTCDATETIME()), 0, '192.168.1.101'),
63 ('Auditor',         DATEADD(HOUR, -1, SYSUTCDATETIME()), 1, '10.10.10.5');
64 GO
65
66
67 EXEC dbo.sp_TransferDistributed
68      @FromAccId = 1,
69      @ToAccId   = 3,
70      @Amount    = 100;
71 GO
72
73
74 EXEC dbo.sp_TransferDistributed
75      @FromAccId = 2,
76      @ToAccId   = 5,
77      @Amount    = 50;
78 GO
```

Listing 58: Script.PostDeployment.sql (BankCentral – dane początkowe

CreateBranchDatabase.sql (DDL bazy Branch1DB i Branch2DB – struktury analogiczne w obu, różnią się jedynie danymi początkowymi):

```
1 CREATE TABLE dbo.BranchClients (
2     BranchClientId   INT IDENTITY(1,1) PRIMARY KEY,
3     CentralClientId  INT NOT NULL,
4     LocalAccountNo   CHAR(20) NOT NULL UNIQUE,
5     Balance          MONEY NOT NULL,
6     LastSync         DATETIME2 NOT NULL DEFAULT SYSUTCDATETIME()
7 );
```

Listing 59: Tables/BranchClients.sql (Oddział)

```
1 CREATE TABLE dbo.BranchTransactions (
2     BranchTxnId    INT IDENTITY(1,1) PRIMARY KEY,
3     BranchClientId INT NOT NULL,
4     Amount         MONEY NOT NULL,
```

```
    TxnType         CHAR(10) NOT NULL CHECK (TxnType IN ('DEBIT','CREDIT')),
    CreatedAt       DATETIME2 NOT NULL DEFAULT SYSUTCDATETIME(),
    Description     NVARCHAR(200) NULL,
    CONSTRAINT FK_BT_BranchClients FOREIGN KEY(BranchClientId) REFERENCES dbo.BranchClients(
    BranchClientId)
);

GO

CREATE INDEX IX_BranchTxn_ClientId ON dbo.BranchTransactions(BranchClientId);
```

Listing 60: Tables/BranchTransactions.sql (Oddział)

```
CREATE TABLE dbo.History (
    HistoryId INT IDENTITY(1,1) PRIMARY KEY,
    Info      NVARCHAR(200) NOT NULL,
    CreatedAt DATETIME2 NOT NULL DEFAULT SYSUTCDATETIME()
);
GO
CREATE INDEX IX_History_CreatedAt ON dbo.History(CreatedAt);
```

Listing 61: Tables/History.sql (Oddział)

```
CREATE VIEW dbo.vBranchClientBalances
AS
SELECT
    bc.BranchClientId,
    bc.LocalAccountNo,
    bc.Balance,
    MAX(bt.CreatedAt) AS LastTxn
FROM dbo.BranchClients bc
LEFT JOIN dbo.BranchTransactions bt ON bc.BranchClientId = bt.BranchClientId
GROUP BY bc.BranchClientId, bc.LocalAccountNo, bc.Balance;
```

Listing 62: Views/BranchClientBalances.sql (Oddział)

```
   /*
Post-Deployment Script Template
--------------------------------------------------------------------------------------
 This file contains SQL statements that will be appended to the build script.
 Use SQLCMD syntax to include a file in the post-deployment script.
 Example:      :r .\myfile.sql
 Use SQLCMD syntax to reference a variable in the post-deployment script.
 Example:      :setvar TableName MyTable
               SELECT * FROM [$(TableName)]
--------------------------------------------------------------------------------------
*/

-- === Klienci lokalni (z centrali BranchId = 1) ===
INSERT INTO dbo.BranchClients (CentralClientId, LocalAccountNo, Balance, LastSync)
SELECT CustomerId, 'BR1_' + RIGHT(AccountNumber, 4), Balance, SYSUTCDATETIME()
FROM BankCentral.dbo.Accounts
WHERE BranchId = 1;
GO

-- === Transakcje lokalne ===
INSERT INTO dbo.BranchTransactions (BranchClientId, Amount, TxnType, Description, CreatedAt)
SELECT bc.BranchClientId, 150.00, 'CREDIT', 'Wpłata bankomatowa', DATEADD(DAY, -2, SYSUTCDATETIME())
FROM dbo.BranchClients bc;
GO

-- === Historia ===
INSERT INTO dbo.History (Info)
VALUES
('Synchronizacja kont lokalnych zakończona sukcesem'),
('Dodano nowe transakcje klientów z centrali');
GO
```

Listing 63: Script.PostDeployment.sql (Branch1DB – inicjalizacja danymi lokalnymi)

```
   /*
Post-Deployment Script Template
--------------------------------------------------------------------------------------
 This file contains SQL statements that will be appended to the build script.
 Use SQLCMD syntax to include a file in the post-deployment script.
 Example:      :r .\myfile.sql
 Use SQLCMD syntax to reference a variable in the post-deployment script.
 Example:      :setvar TableName MyTable
               SELECT * FROM [$(TableName)]
```

```
--------------------------------------------------------------------------------
*/

-- === Klienci lokalni (z centrali BranchId = 2) ===
INSERT INTO dbo.BranchClients (CentralClientId, LocalAccountNo, Balance, LastSync)
SELECT CustomerId, 'BR2_' + RIGHT(AccountNumber, 4), Balance, SYSUTCDATETIME()
FROM BankCentral.dbo.Accounts
WHERE BranchId = 2;
GO

-- === Transakcje lokalne ===
INSERT INTO dbo.BranchTransactions (BranchClientId, Amount, TxnType, Description, CreatedAt)
SELECT bc.BranchClientId, 75.00, 'DEBIT', 'Opłata lokalna', DATEADD(DAY, -1, SYSUTCDATETIME())
FROM dbo.BranchClients bc;
GO

-- === Historia ===
INSERT INTO dbo.History (Info)
VALUES
('Dane klientów z centrali zsynchronizowane'),
('Utworzono transakcje lokalne na podstawie danych centralnych');
GO
```

Listing 64: Script.PostDeployment.sql (Branch2DB – inicjalizacja

E. Kod warstwy testowej

Tests.cs – przykładowy skrypt testujący procedury (alternatywnie do testów w C#):