Webowa Aplikacja Bankowa

Autorzy: Witold Pacholik, Wojciech Grzywocz, Piotr Olasik

Opis Projektu

Projekt implementuje prostą aplikację bankową z podstawowymi funkcjonalnościami, zrealizowaną przy użyciu technologii **C# ASP.NET Core MVC z Razor Views**.

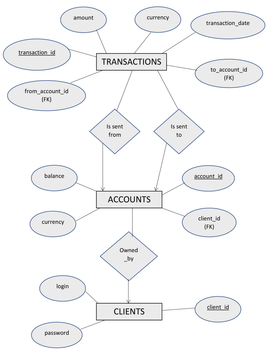
Kluczowe Funkcjonalności Projektu

- Rejestracja i logowanie użytkowników
- Możliwość tworzenia, przeglądania i zarządzania kontami bankowymi
- Transakcje pomiędzy kontami (wypłaty, przelewy)
- Przeglądanie historii transakcji
- Przeliczanie walut podczas przelewów z wykorzystaniem aktualnych kursów z zewnętrznego API
- Uwierzytelnianie oparte na ciasteczkach (cookie-based authentication)

Schemat Bazy Danych (PostgreSQL hostowany na Railway)

Poniżej znajduje się schemat bazy danych utworzony w PostgreSQL i hostowany na platformie Railway, definujący strukturę tabel dla użytkowników, kont i transakcji.

Diagram Związków Encji



Opis Bazy Danych

Baza danych aplikacji została zaprojektowana przy użyciu **Entity Framework Core** jako ORM i oparta na **PostgreSQL**. Służy do zarządzania użytkownikami, kontami bankowymi oraz transakcjami między kontami. Baza danych była tworzona automatycznie przy pomocy migracji EF Core.

Relacje

- Jeden klient może posiadać wiele kont bankowych (1:N).
- Transakcja zawiera referencje do dwóch kont: źródłowego i docelowego (w implementacji ignorowane w EF, lecz logicznie są to relacje N:1).

Inne Cechy

- `CURRENCY` to enum .NET – może być rozszerzony do mapowania obiektów typu string
- `LOGIN` jest indeksowany, co oznacza, że baza danych może szybko wyszukiwać klientów po tej kolumnie (np. podczas logowania) i unikalny – nie mogą istnieć dwa rekordy z tą samą wartością, co zapewnia integralność danych przy rejestracji.
- Hasła są przechowywane jako hash funkcją bcrypt

Zachowanie relacji i usuwanie powiązanych transakcji

W projekcie przyjęliśmy specyficzne podejście do odwzorowania relacji między kontami bankowymi a transakcjami. Transakcje mają dwa pola identyfikujące konta uczestniczące w operacji: `FromAccountId` oraz `ToAccountId`. Każda transakcja reprezentuje albo:

- **Wypłatę** – tylko `FromAccountId` jest ustawione. `ToAccountId` pozostaje `null`.
- **Przelew** – oba pola `FromAccountId` i `ToAccountId` są wypełnione.

W modelu danych (`ApplicationDbContext`) zdecydowaliśmy się **pominąć relacje nawigacyjne** `FromAccount` i `ToAccount` , przy użyciu:

```
entity.Ignore(t => t.FromAccount);
entity.Ignore(t => t.ToAccount);
```

Dlaczego pominęliśmy relacje nawigacyjne?

- **Aby uniknąć cyklicznych zależności w EF Core** – relacje wiele-do-wielu mogłyby prowadzić do złożonych cykli.
- **Aby poprawić wydajność** – tworzenie transakcji z wieloma JOIN-ami mogłoby spowalniać aplikację.
- **Aby zwiększyć kontrole nad kaskadowym usuwaniem** – brak relacji nawigacyjnych pozwala lepiej zarządzać logiką usuwania.
- **Ręczne zapytania ułatwiają też obsługę sytuacji wyjątkowych**, np. gdy ToAccountId jest null (dzięki temu unikamy sytuacji, gdzie ORM próbuje załadować „puste" powiązanie i generuje niepotrzebne lub błędne zapytania).

Skutek przy usuwaniu kont

- **Transakcje wypłat** (`ToAccountId == null`) są usuwane automatycznie, gdy konto źródłowe (`FromAccountId`) zostanie usunięte.
- **Transakcje przelewów** (oba pola niepuste) pozostają, dopóki oba konta nie zostaną usunięte.

Tzn.:

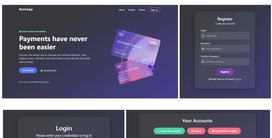
- Usunięcie jednego konta – transakcja nadal istnieje i wskazuje na drugie konto.
- Dopiero po usunięciu obu kont – transakcja jest usuwana.

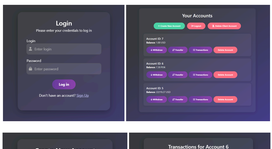
Jak to osiągnęliśmy?

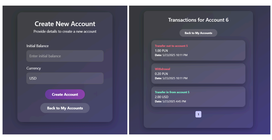
- Nie zdefiniowaliśmy `HasOne()` / `WithMany()` między `Transaction a Account` → brak kaskadowego usuwania i relacji nawigacyjnych.
- Pola `FromAccountId` i `ToAccountId` są traktowane jako zwykłe klucze obce – bez relacji w C#.

Takie rozwiązanie jest bardziej elastyczne i wydajne w prostej aplikacji.

Prezentacja Interfejsu Aplikacji







Technologie Wykorzystane w Warstwie Bazy Danych

- Entity Framework Core (ORM i migracje schematu bazy danych)
- PostgreSQL hostowany na Railway
- JetBrains Rider z database navigator plugin
- Postman (narzędzie wykorzystywane do testowania zapytań do bazy danych / API)

Podsumowanie Projektu

Zaprojektowana baza danych wspiera podstawowe operacje bankowe w aplikacji i jest zgodna z założeniami funkcjonalnymi. Struktura bazy jest prosta, a jednocześnie elastyczna pod kątem dalszego rozwoju.

Potencjał Rozwoju

W przyszłości planujemy rozbudowę bazy danych o nowe funkcjonalności:

- Lazy loading relacji z kontami (opcjonalnie) - Obecnie relacje nawigacyjne FromAccount i ToAccount w encji Transaction są ignorowane w konfiguracji Entity Framework Core, co oznacza, że aplikacja nie wykorzystuje relacji ani zapytań typu JOIN do pobierania danych powiązanych kont. To podejście zwiększa wydajność i upraszcza zarządzanie relacjami oraz usuwaniem danych. W przyszłości planujemy możliwość włączenia lazy loadingu, który umożliwi automatyczne pobieranie danych powiązanych kont bez konieczności ręcznego wykonywania zapytań. Taki mechanizm może być użyteczny przy rozbudowie interfejsu użytkownika lub API, gdzie potrzebne są szczegółowe dane transakcji z informacjami o kontach.
- Kategorie transakcji i opisy - Dodanie tabeli TransactionCategory i powiązania jej z Transaction, co umożliwi klasyfikację wydatków (np. żywność, transport, rozrywka) oraz dodanie opisu do każdej transakcji.
- Historia powiadomień - Utworzenie tabeli NotificationLog przechowującej historię wysyłanych powiadomień e-mailowych lub SMS-ów (czas, treść, status, powiązany użytkownik).
- Uwierzytelnianie dwuetapowe (2FA) - Rozszerzenie encji Client o kolumny związane z 2FA, np. TwoFactorEnabled, TwoFactorSecret, Last2FAVerification.
- Eksport historii transakcji - Dodanie tabeli ExportRequest, przechowującej informacje o żądaniach eksportu (format, zakres dat, klient, status wykonania).
- REST API i logowanie zapytań - Utworzenie tabeli ApiAccessLog, rejestrującej wywołania API (endpoint, czas, użytkownik, IP), co wspiera analizę i bezpieczeństwo.
- Archiwizacja danych - Wprowadzenie mechanizmu archiwizacji starych transakcji do oddzielnej tabeli lub bazy (np. ArchivedTransactions) w celu zwiększenia wydajności operacyjnej.