**DOKUMENTACJA PROJEKTU**  
Bazy Danych II

Przetwarzanie rozproszone w SQL Server  
Radosław Porębiński

1. **Opis problemu i opis funkcjonalności udostępnianej przez API**

W ramach projektu stworzony został system symulujący rozproszony system bankowy z centralną bazą klientów oraz oddziałami banku posiadającymi własne lokalne bazy danych, w których dodatkowo rejestrowane są transakcje bankowe. Celem projektu było opracowanie biblioteki umożliwiającej komunikację z rozproszonymi bazami danych SQL Server i zarządzanie transakcjami. Założeniem dodatkowym były, że bazy danych nie wiedzą o sobie nawzajem, wtedy dla utrudnienia, całe przetwarzanie rozproszone realizujemy z wykorzystaniem mikroserwisów. Do realizacji zostały wykorzystane technologie:

* Spring Boot 3
* SQL Server
* Angular 19
* Docker
* Apache Kafka

API umożliwia:

* Stworzenie kont logowania
* Stworzenie poszczególnych kont bankowych przypisanych do kont logowania
* Wykonywanie przelewu między dwoma kontami
* Wykonanie wpłaty lub wypłaty
* Wygenerowanie nowej karty kredytowej
* Wykonanie transakcji symulującej płatność kartą np. w sklepie
* Oraz operacje na tych danych takie jak edytowanie, wyświetlanie i usuwanie

1. **Opis typów danych oraz metod (funkcji) udostępnionych w ramach API**

Konta logowania:

**public class User {**

**@Id**

**private Integer id;**

**private String username;**

**private String password;**

**private String email;**

**private String avatar;**

**private Role role;**

**private LocalDateTime createdAt;**

**}**

Rola użytkownika:

**public enum Role {**

***USER*,**

***ADMIN***

**}**

Konto bankowe:

**public class Client {**

**@Id**

**private UUID id;**

**private ClientStatus status;**

**private String firstName;**

**private String lastName;**

**private int userId;**

**private String phone;**

**private String address;**

**private String city;**

**private int branch;**

**@Column(unique = true)**

**private String accountNumber;**

**private double balance;**

**private double balanceReserved;**

**}**

Status użytkownika:

**public enum ClientStatus {**

***CREATING*,**

***ACTIVE***

**}**

Przelew:

**public class Transfer {**

**@Id**

**private UUID id;**

**private String fromAccount;**

**private String toAccount;**

**private double amount;**

**private TransferStatus status;**

**private int fromBranchId;**

**private int toBranchId;**

**private long date;**

**}**

Status przelewu:

**public enum TransferStatus {**

***STARTED*,**

***COMPLETED*,**

***FAILED***

**}**

Karta kredytowa:

**public class CreditCard {**

**@Id**

**private String cardNumber;**

**private String accountNumber;**

**private String expirationDate;**

**private String cvv;**

**}**

Operacja wpłaty/wypłaty:

**public class BalanceChange {**

**@Id**

**private UUID id;**

**private String account;**

**private double amount;**

**private BalanceChangeStatus status;**

**private int branchId;**

**private long date;**

**}**

Status wpłaty/wypłaty:

**public enum BalanceChangeStatus {**

***STARTED*,**

***COMPLETED***

**}**

**API w naszym przypadku to REST API:**

Do poprawnej obsługi mikroserwisów, musiało zostać udostępnione więcej metod niż wykorzystuje np. Przykładowy interfejs użytkownika

# **AuthService - AuthController**

ResponseEntity<**UserDTO**>

**createUser**(@Valid **UserDTO** user)

ResponseEntity<**JWTAuthentication**>

**loginUser**(@Valid **LoginDTO** user)

ResponseEntity<**JWTAuthentication**>

**refreshToken**(@Valid **JWTAuthentication** authentication)

ResponseEntity<**UserDTO**>

**updateUser**(@Valid **UserUpdateDTO** user, int userId)

ResponseEntity<**UserDTO**>

**validateToken**(@Valid **JWTAuthentication** authentication)

# **BranchService - ClientController**

**ResponseEntity<Client>**

**create(@Valid Client client, int userId)**

**ResponseEntity<CreditCard>**

**createCreditCard(**[**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **account)**

**ResponseEntity<Void>**

**deleteCreditCard(**[**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **number)**

**ResponseEntity<Client>**

**getClient(**[**UUID**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/UUID.html) **id**

**ResponseEntity<Client>**

**update(@Valid Client client,** [**UUID**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/UUID.html) **id, int userId)**

# **BranchService - PaymentController**

**ResponseEntity<BalanceChange>**

**deposit(**[**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **account, double amount)**

**ResponseEntity<BalanceChange>**

**getBalance(**[**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **account,** [**UUID**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/UUID.html) **id)**

**ResponseEntity<**[**List**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html)**<BalanceChange>>**

**getBalanceChangesByAccount(**[**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **account)**

**ResponseEntity<Transfer>**

**getTransfer(**[**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **account,** [**UUID**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/UUID.html) **id)**

**ResponseEntity<**[**List**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html)**<Transfer>>**

**getTransfersByAccount(**[**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **account)**

**ResponseEntity<Transfer>**

**makePaymentTransfer(**[**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **from,** [**UUID**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/UUID.html) **to, double amount)**

**ResponseEntity<Transfer>**

**makeTransfer(**[**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **from,** [**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **to, double amount)**

**ResponseEntity<BalanceChange>**

**withdraw(**[**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **account, double amount)**

# **CreditCardService - CreditCardController**

**ResponseEntity**<**Bank**>

**createBank**(**Bank** **bank**)

**ResponseEntity**<**CreditCard**>

**createCreditCard**(int **bank**, [**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **account**)

**ResponseEntity**<[**Void**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Void.html)>

**deleteCreditCard**([**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **number,** int **userId**)

**ResponseEntity**<[**List**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html)<**CreditCard**>>

**getCreditCards**(int **userId**)

**ResponseEntity**<**Transfer**>

**pay**([**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **number**, [**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **date**, [**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **cvv**, [**UUID**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/UUID.html) **service**, double **amount**)

# **HeadquarterService - ClientController**

**ResponseEntity**<[**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html)>

**getAccountNumber**([**UUID**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/UUID.html) **id**)

**ResponseEntity**<[**List**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html)<**Client**>>

**getAccounts**(int **id**)

**ResponseEntity**<[**List**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html)<[**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html)>>

**getAccountsByUserId**(int **id**)

**ResponseEntity**<[**Integer**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Integer.html)>

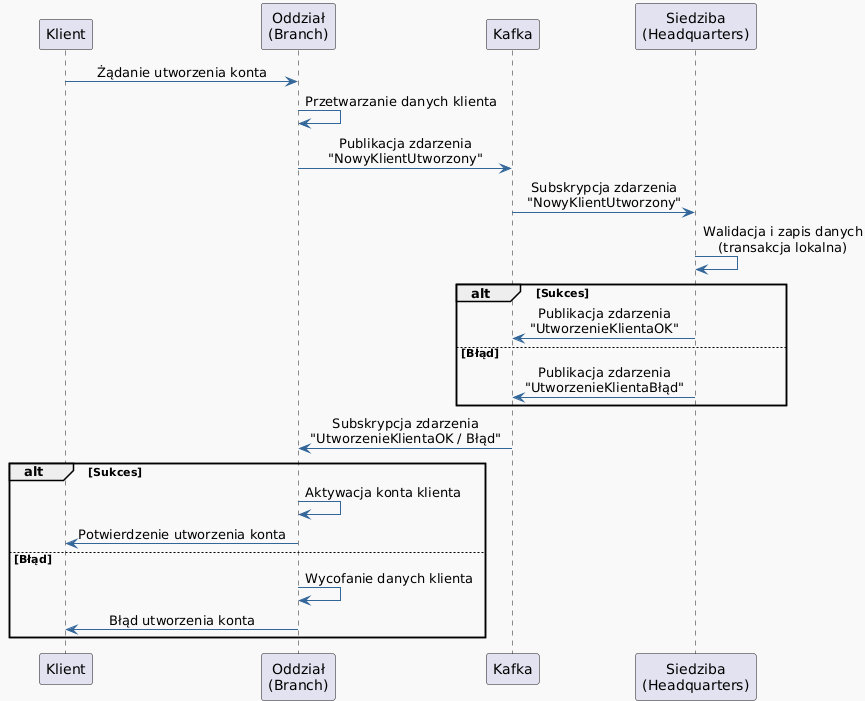
**getBranchId**([**String**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html) **account**)

1. **Opis implementacji udostępnionego API przez bibliotekę**

API zostało zrealizowane z wykorzystaniem **mikroserwisów**. Do implementacji transakcji rozproszonych został wykorzystany **SAGA Pattern** ze sposobem komunikacji opartym o **Choreografię**. Aby poprawnie obsłużyć transakcje rozproszone musimy wykorzystać kompensację zdarzeń.

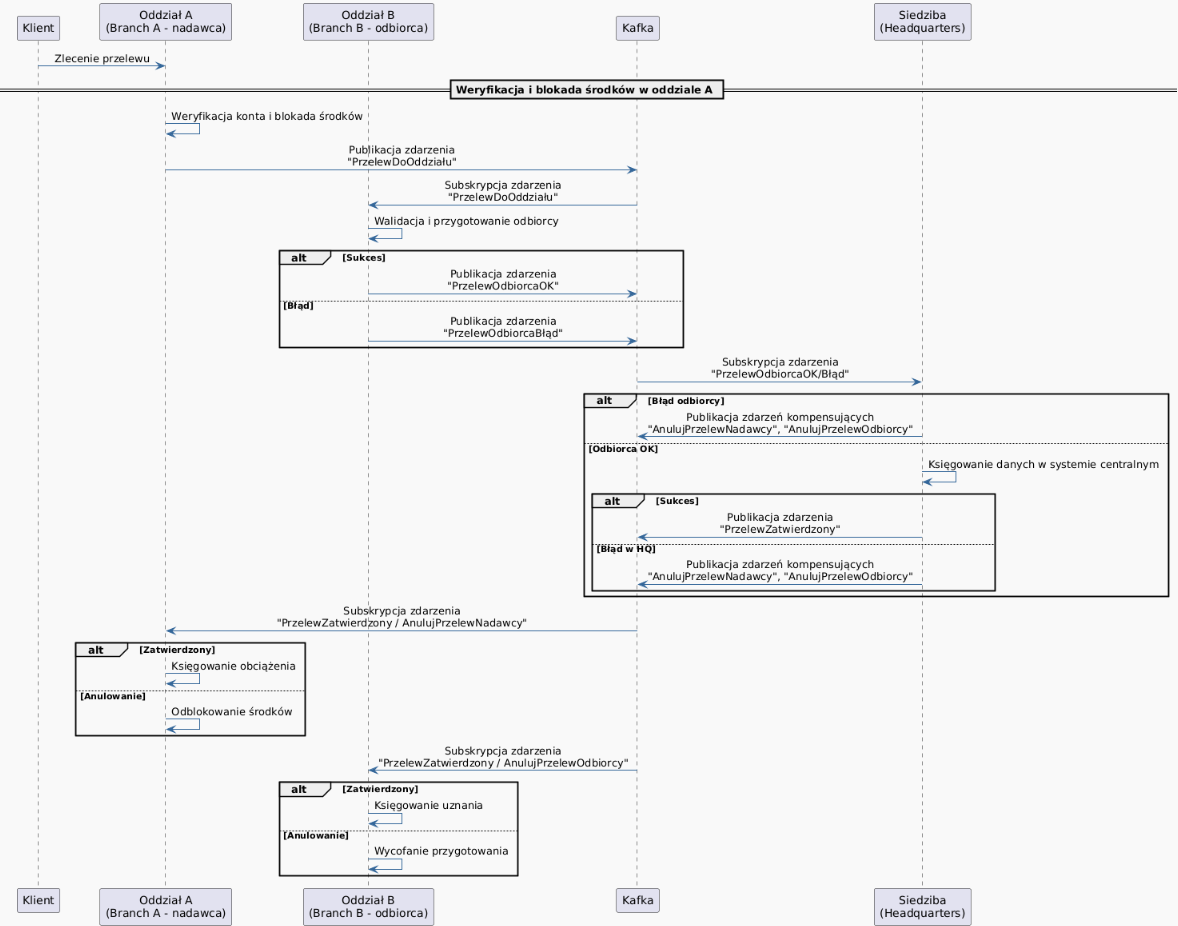
Tworzenie konta bankowego przebiega w krokach:

* Wysłanie zapytania do danego branchu i przetworzenie przez niego danych
* Branch wysyła zdarzenie do siedziby o utworzeniu nowego klienta
* Siedziba odbiera zdarzenie i przetwarza i zapisuje dane, wysyła zdarzenie powrotne o poprawnym utworzeniu klienta, jeżeli wystąpił błąd, dane nie są zapisywane (lokalna transakcja), również jest wysyłane zdarzenie powrotne
* Branch w zależności od zdarzenia, aktywuje klienta lub cofa jego utworzenie



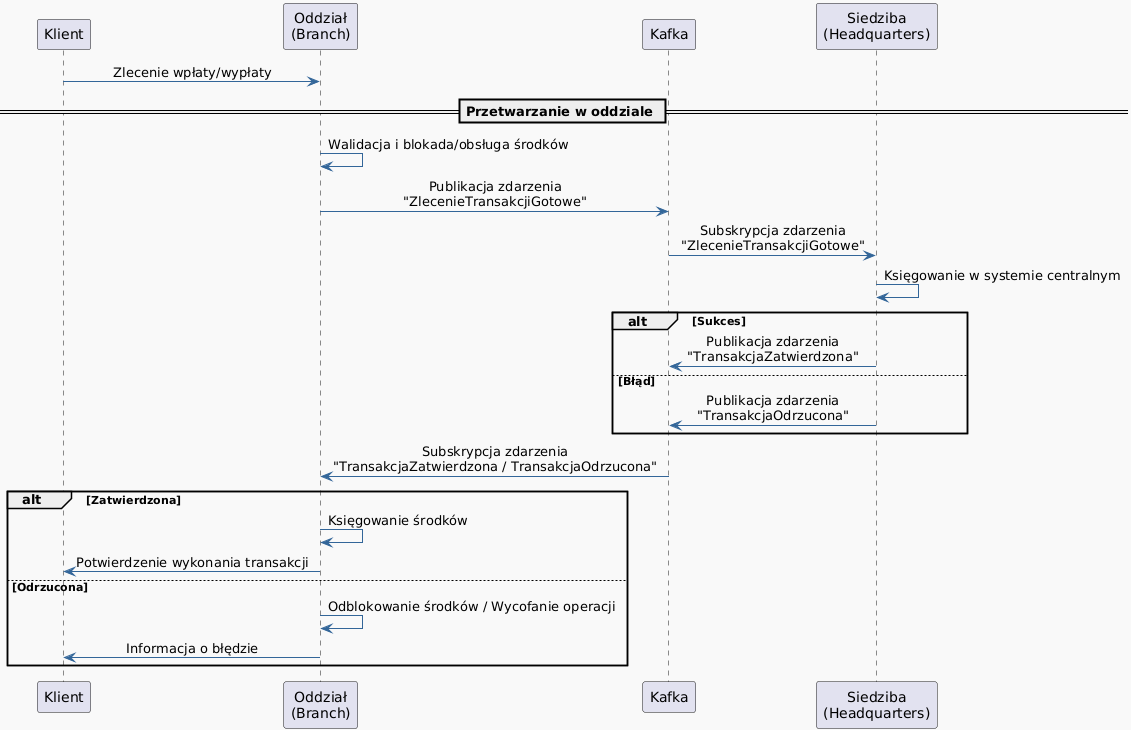
Przelew między dwoma kontami bankowymi przebiega w krokach:

* Wysłanie zapytania do danego branchu i przetworzenie przez niego danych, w tym zablokowanie potrzebnych środków
* Branch wysyła zdarzenie do drugiego branchu, lub do siedziby, jeżeli jest to transakcje wewnątrz tego oddziału
* Drugi branch również przetwarza dane, jeżeli wystąpi błąd dane nie są zapisywane (lokalna transakcja), ponownie wysyłamy zdarzenie powrotne
* Jeżeli dane przez drugi branch zostały zapisane pomyślnie, wysyła on zdarzenie do siedziby
* Siedziba odbiera zdarzenie i przetwarza i zapisuje dane, jeżeli wystąpi błąd, musimy wysłać zdarzenie kompensujące do obydwóch branchy. Jeżeli transakcja została zrealizowana pomyślnie, wysyłamy zdarzenie do branchy
* Branche odbierają zdarzenie i księgują środki na kontach, lub w przypadku niepowodzenia odblokowują środki



Wpłata lub wypłata środków przebiega w krokach:

* Wysłanie zapytania do danego branchu i przetworzenie przez niego danych, w tym zablokowanie potrzebnych środków
* Branch wysyła zdarzenie do siedziby
* Siedziba odbiera zdarzenie i przetwarza i zapisuje dane, jeżeli wystąpi błąd, musimy wysłać zdarzenie kompensujące do brancha. Jeżeli transakcja została zrealizowana pomyślnie, wysyłamy zdarzenie do brancha.
* Branch odbiera zdarzenie i księguje środki na koncie, lub w przypadku niepowodzenia odblokowuje środki.



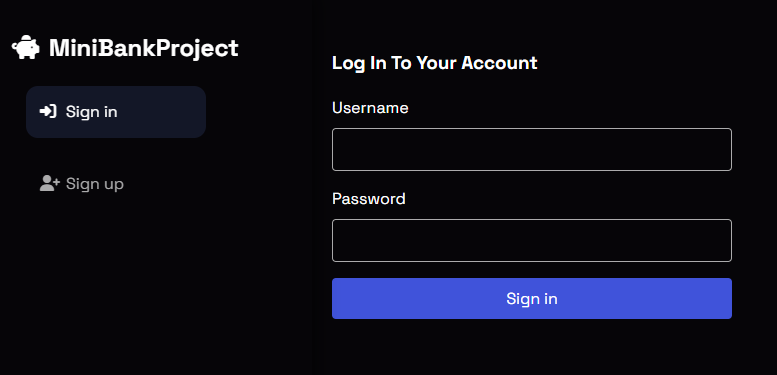
Warto zauważyć, że we wszystkich przypadkach, w przypadku przelania z danego konta, od razu musimy zabezpieczyć środki, a przypadku wpłaty na konto, tych środków nie możemy od razu przydzielić - aby nie zostały wykorzystane w między czasie, a transakcja może się w tym czasie nie powieść.

1. **Prezentacja przeprowadzonych testów jednostkowych**

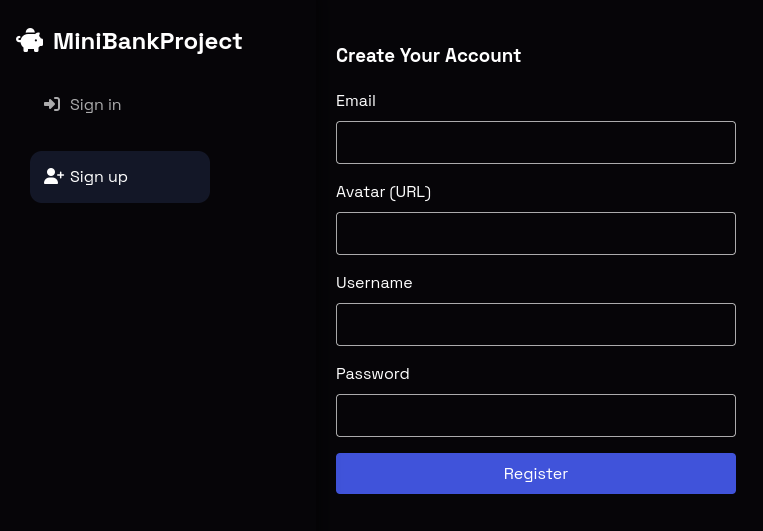
[Załącznik: testy.zip]

1. **Prezentacja przykładowej aplikacji**

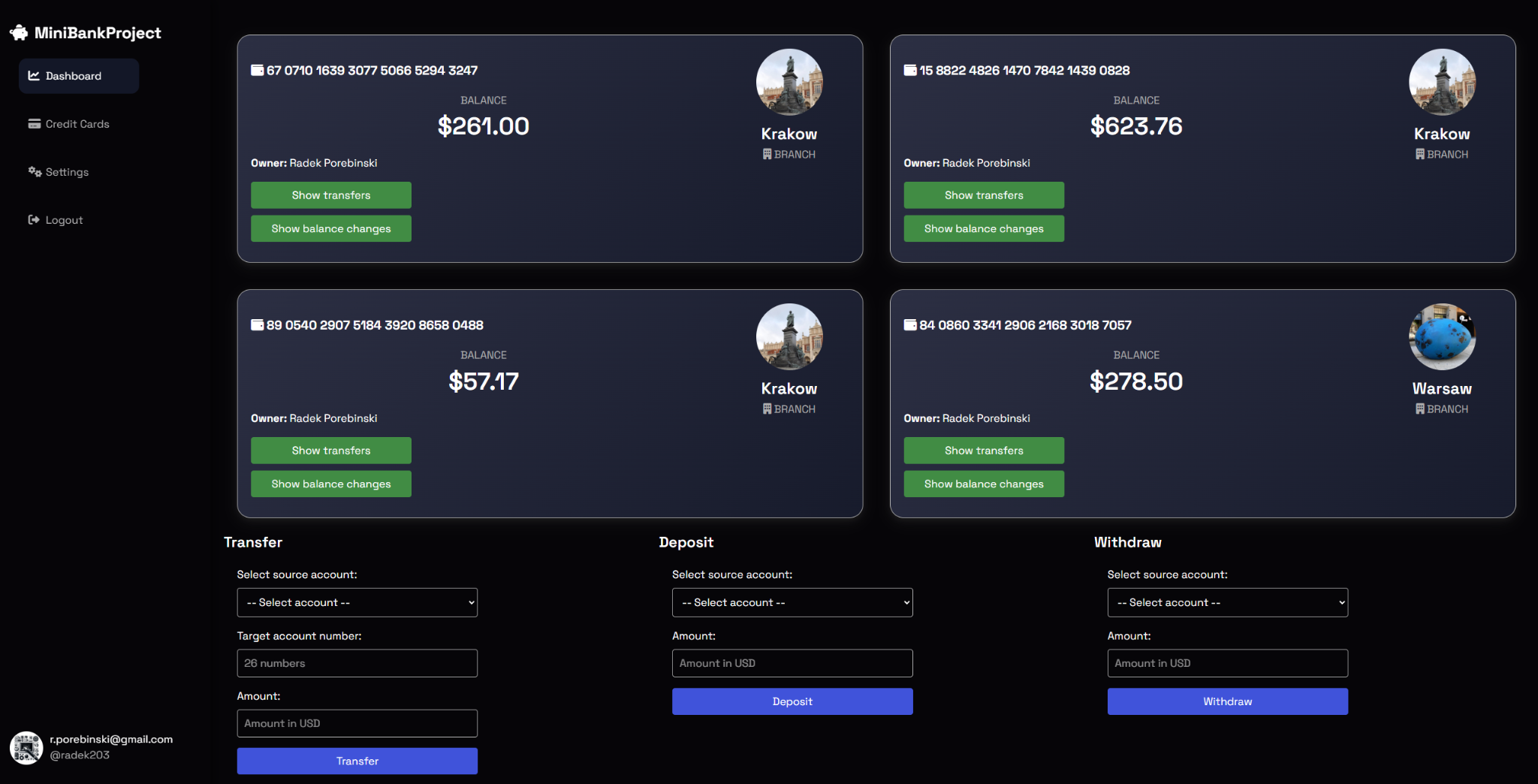
Ekran logowania:

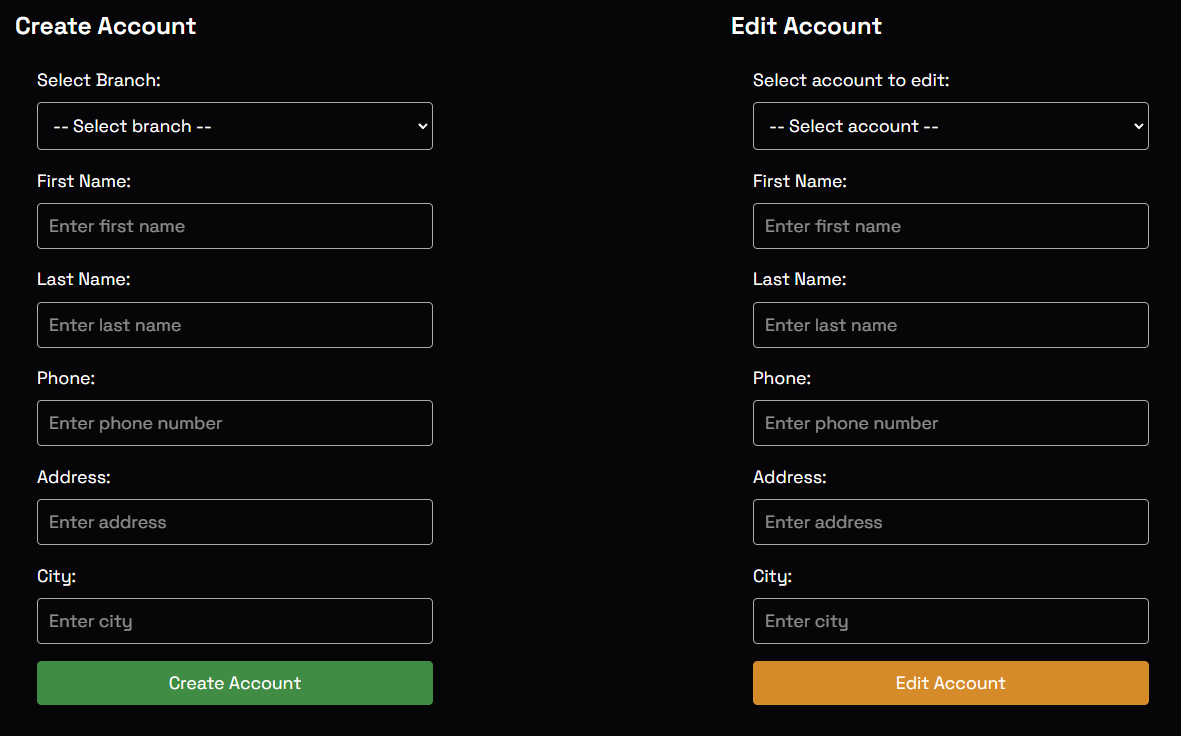
****

Ekran rejestracji:

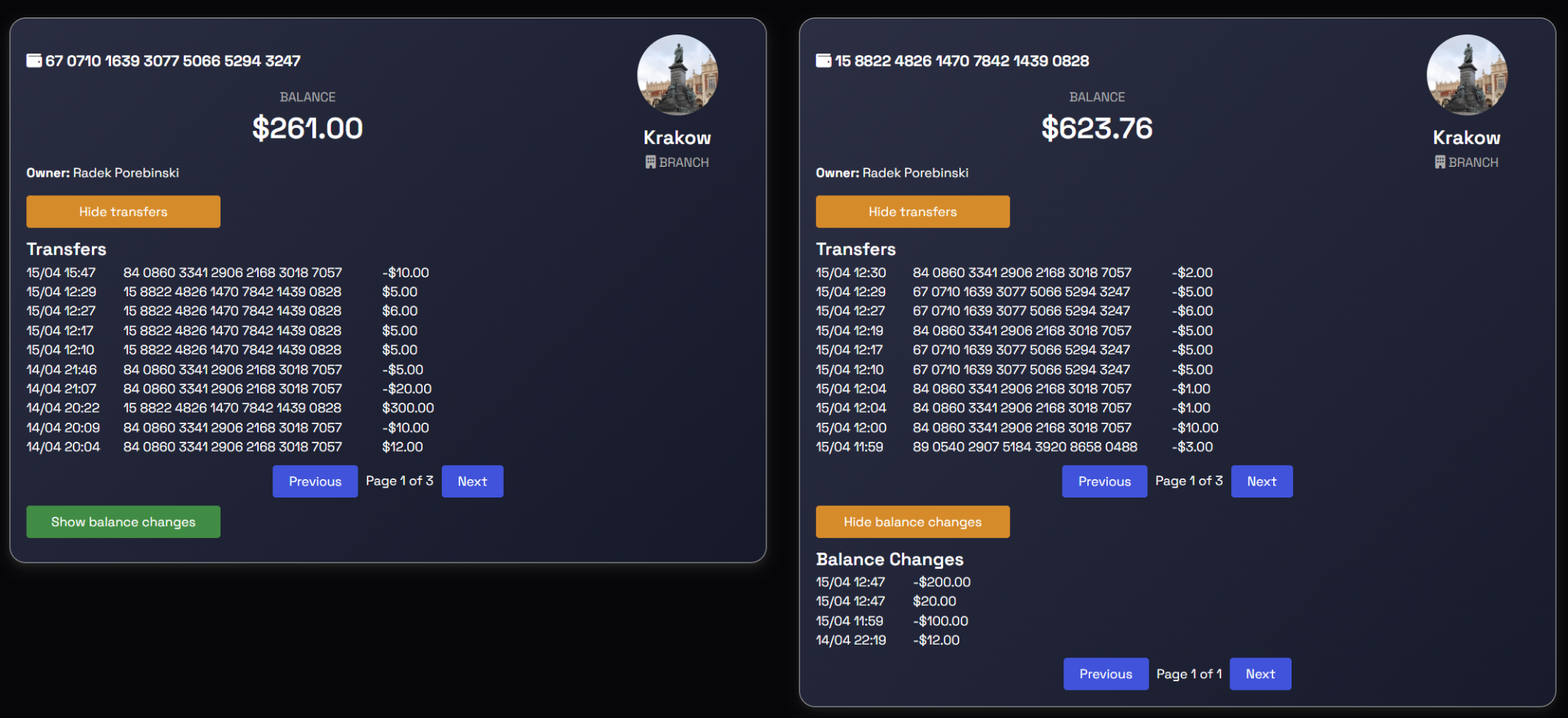
****

Panel główny:

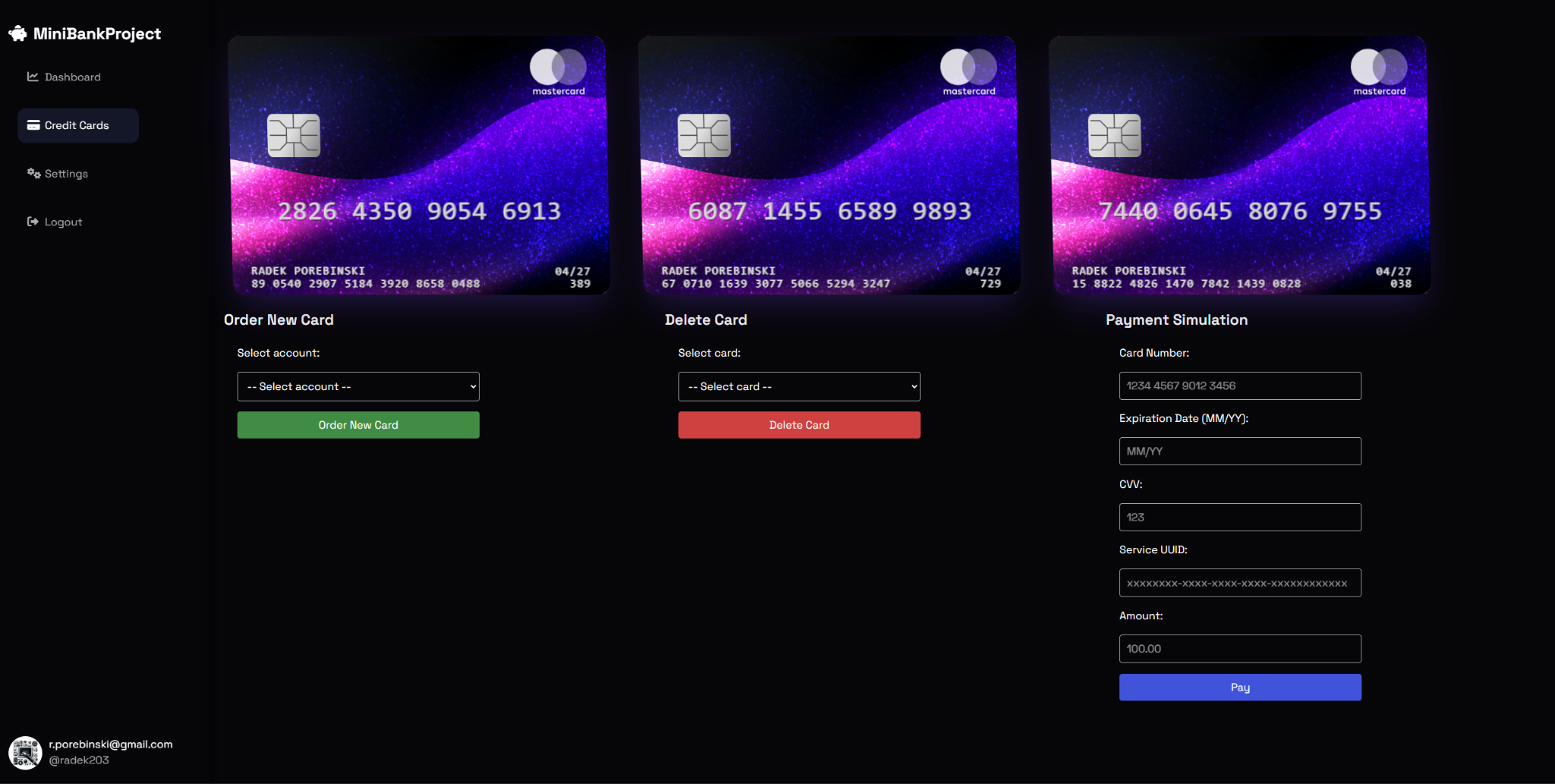
****

****

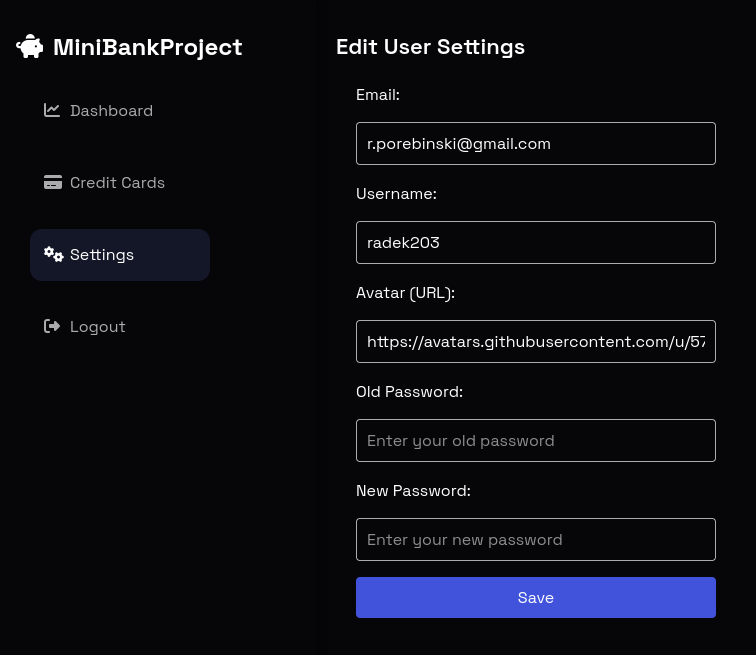
Podgląd wykonanych transakcji:

****

Ekran z kartami kredytowymi:

****

Ekran edytowania użytkownika:

****

1. **Podsumowanie i wnioski**

Projekt miał na celu zaprojektowanie i implementację systemu rozproszonego do obsługi operacji bankowych z wykorzystaniem baz danych MS SQL Server oraz mikroserwisowej architektury komunikującej się poprzez REST API. Centralnym elementem systemu była biblioteka umożliwiająca obsługę transakcji rozproszonych, zgodnie ze wzorcem SAGA, przy zastosowaniu choreografii zdarzeń.

W ramach projektu stworzono kompletne API wspierające kluczowe operacje bankowe, takie jak rejestracja użytkownika, zakładanie kont, przelewy, wpłaty, wypłaty oraz płatności kartą kredytową. System pozwala również na zarządzanie danymi użytkownika i kont bankowych, a także na pełną obsługę cyklu życia transakcji – od rozpoczęcia, przez rezerwację środków, aż po ich zaksięgowanie lub cofnięcie w przypadku błędów.

Ważnym aspektem implementacji było zapewnienie spójności danych w środowisku rozproszonym, co zrealizowano poprzez wykorzystanie zdarzeń kompensacyjnych w przypadku niepowodzenia któregokolwiek etapu transakcji. Dzięki temu każda operacja rozproszona została zabezpieczona mechanizmami przywracającymi stan systemu do momentu sprzed rozpoczęcia transakcji, zapewniając integralność danych i bezpieczeństwo operacji.

System został przetestowany zarówno na poziomie jednostkowym, a także zaprezentowany za pomocą prostego interfejsu użytkownika, pozwalającego na weryfikację działania poszczególnych funkcji.

#### Wnioski:

* Wykorzystanie wzorca SAGA z choreografią okazało się efektywnym podejściem do obsługi transakcji rozproszonych w środowisku mikrousług.
* Implementacja zdarzeń kompensacyjnych jest kluczowa dla zapewnienia spójności danych w przypadku błędów podczas przetwarzania transakcji.
* MS SQL Server umożliwia skuteczne zarządzanie danymi w architekturze rozproszonej, choć wymaga dodatkowych warstw logiki przy projektowaniu mechanizmów synchronizacji.
* Mikroserwisy, mimo że zwiększają złożoność projektu, pozwalają na elastyczne skalowanie i łatwiejsze zarządzanie poszczególnymi komponentami systemu.
* Projekt stanowi solidną podstawę do dalszego rozwoju, np. poprzez integrację z zewnętrznymi usługami płatniczymi, rozszerzenie funkcjonalności kart kredytowych lub wdrożenie systemu notyfikacji transakcji w czasie rzeczywistym.

W trakcie realizacji projektu zauważono, że implementacja mikroserwisów, mimo że zapewnia wysoką skalowalność i separację odpowiedzialności, wiąże się z dużym poziomem złożoności. Zarządzanie komunikacją między serwisami, obsługa błędów, spójność danych i zapewnienie niezawodności wymagały dodatkowego nakładu pracy oraz zastosowania zaawansowanych technik. W szczególności integracja rozproszonych baz danych wymagała dobrze przemyślanej logiki obsługi zdarzeń i kompensacji.

W związku z tym należy zaznaczyć, że **mikroserwisy nie zawsze są najlepszym wyborem architektonicznym**. Jeśli system nie wymaga wysokiej skalowalności, niezależnego wdrażania modułów ani dużej elastyczności w rozwoju, to **zdecydowanie lepszym i prostszym podejściem może być zastosowanie monolitu**. Mikroserwisy generują dodatkowe koszty – zarówno implementacyjne, jak i infrastrukturalne – dlatego ich użycie powinno być świadomą decyzją, popartą rzeczywistymi wymaganiami projektowymi.

Projekt pozwolił na praktyczne poznanie i zastosowanie zaawansowanych koncepcji z zakresu przetwarzania rozproszonego, a także ukazał wyzwania związane z realną implementacją architektury mikroserwisowej.

1. **Literatura**

* Dokumentacja Spring Boot 3: [**https://spring.io/projects/spring-boot**](https://spring.io/projects/spring-boot)
* Dokumentacja Angular 19:[**https://angular.dev**](https://angular.dev/)
* Dokumentacja MS SQL Server: [**https://learn.microsoft.com/en-us/sql/?view=sql-server-ver16**](https://learn.microsoft.com/en-us/sql/?view=sql-server-ver16)
* O Transakcjach rozproszonych i mikroserwisach: [**https://medium.com/**](https://medium.com/)