

Cel projektu

Głównym celem projektu było stworzenie aplikacji, która będzie symulowała działanie banku internetowego. Najbardziej zależało mi na stworzeniu funkcjonalności wykonywania przelewów. Kolejnym wymaganiem było, aby nie była to aplikacja lokalna lecz globalna, to znaczy umożliwiającą wymianę danych pomiędzy różnymi urządzeniami. Elementem koniecznym w działaniu banku jest to, że działania dokonane przez jednego klienta mają wpływ na innych. Dokonując przelewu ważne jest przecież, aby pieniądze wysłane z jednego konta pojawiły się na drugim. Aby zasumulować ten proces użyłem bazy danych hostowanej w chmurze. Dzięki takiemu rozwiązaniu każdy kto zainstaluje moją aplikację będzie mieć możliwość zawierania transakcji z innymi użytkownikami. Inne funkcjonalności jakie udostępnia mój program to tworzenie i autentykacja użytkowników; oraz tworzenie, usuwanie, edytowanie kont. Zatem każdy możliwy typ operacji CRUD jest obsługiwany. Podstawową zasadą jest to, że użytkownik ma dostęp tylko do swoich kont oraz do historii tylko i wyłącznie swoich transakcji.

Jako cel programistyczny postawiłem sobie uzyskanie jak największej modułowości programu. Chciałem, aby nowe moduły były łatwe do dodania oraz nie powodowały problemów w stosunku do obecnych już funkcjonalności.

Domena

Jeśli chodzi o domenę biznesową aplikacji bankowej, to zaliczyłbym do niej konta, transakcje i użytkowników. Pozostałe funkcjonalności są tylko dodatkiem do tych elementów.

Aplikacja bankowa powinna mieć podstawowe cechy takie jak:

- ochrona danych poprzez używanie kont użytkowników z enkrypcją haseł
- przechowywanie danych w bezpiecznej lokalizacji niezależnej od naszego klienta(nie można ufać w bezpieczeństwo komputera na którym uruchamiany jest nasz program)
- spójność danych(podczas przelewu suma pieniędzy pomiędzy dwoma kontami musi pozostać stała)

Wykorzystane narzędzia

Do stworzenia programu używałem następujących technologii:

- Azure SQL Database
- Java ze strukturą projektu Maven
- JavaFX
- Hibernate
- system kontroli wersji GIT

Oraz narzędzi:

- Intellij IDEA
- SceneBuilder
- GitHub
- Azure Data Studio
- JPA Buddy Plugin

Uzasadnienie

Język programowania Java, rozwiązanie do tworzenia GUI- JavaFX oraz ORM Hibernate zostały z góry narzucone w wymaganiach projektu. Natomiast jako moją bazę danych wybrałem rozwiązanie Azure, ponieważ Uniwersytet Rzeszowski udostępnia mi darmową subskrypcję do celów edukacyjnych, a rozwiązania chmurowe poza pakietami edukacyjnymi wymagają podpięcia karty kredytowej, czego starałem się uniknąć, nie chcąc narażać się na nieprzewidziane koszty. Technologia Maven znacznie ułatwia pracę poprzez automatyczne ładowanie dependencji. Używałem githuba, aby stworzyć kopię zapasową mojego projektu oraz aby mieć możliwość przywrócenia wcześniejszej wersji programu w przypadku wystąpienia niespodziewanych błędów lub zmiany koncepcji. Scene builder znacznie ułatwił mi pracę podczas tworzenia GUI, ponieważ uprościł zastosowanie wzorca projektowego MVC (Model widok kontroler). Widoki są ładowane z plików xml, nadaje się im style używając css'a oraz zarządza się nimi za pomocą kontrolerów. IntelliJ IDEA jest moim ulubionym narzędziem do pracy w języku JAVA, ponieważ udostępnia szereg funkcjonalności, które przyspieszają tworzenie

aplikacji. Umożliwia on także używanie dodatków, takich jak na przykład JPA Buddy, dzięki któremu w prosty sposób wygenerowałem modele na podstawie istniejącej bazy danych, które wystarczyło potem tylko nieznacznie dostosować do dalszej pracy. Pracowałem na systemie Windows.

[Jak powstawał projekt?](#)

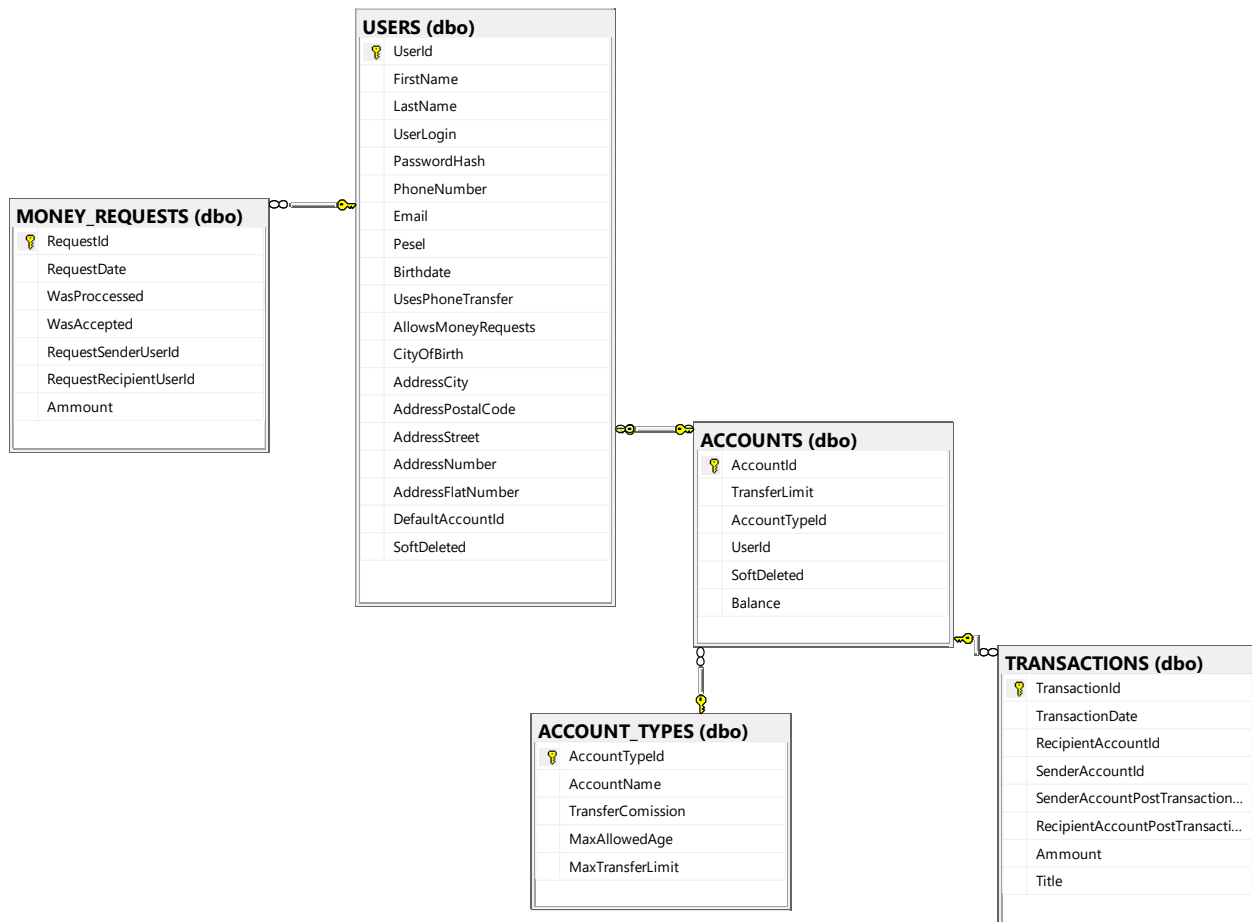
Kierowałem się następującym schematem według kolejności:

1. Zdefiniowanie wymagań aplikacji
2. Określenie modelu danych
3. Stworzenie bazy danych wraz z odpowiednimi więzami integralności oraz relacjami pomiędzy encjami
4. Utworzenie projektu w IDE oraz stworzenie repozytorium na githubie.
5. Uzyskanie połączenia z bazą w tworzonej aplikacji.
6. Stworzenie logiki aplikacji tzw. backendu.
7. Stworzenie GUI (tzw. frontend).

Struktura danych

Do zobrazowania struktury danych najlepiej służy diagram ERD. Opisuje on poszczególne encje oraz relacje między nimi.

Diagram mojej aplikacji:



Najbardziej rozbudowaną tabelą jest tabela **USERS**. Zawiera informacje o konkretnym użytkowniku. Warto zwrócić tutaj szczególną uwagę na dwa pola:

PasswordHash- jest to zhashowane hasło. Zostało poddane enkrypcji przy użyciu techniki hash + salt. Chroni ona przed najczęstszymi technikami używanymi do uzyskania nieuprawnionego dostępu do konta. Opowiem o niej w późniejszej części dokumentacji.

SoftDeleted- ponieważ zdefiniowane więzy integralności uniemożliwiałyby nam usunięcie użytkownika, musimy użyć techniki zwanej soft deleting. Polega ona na przechowywaniu informacji o tym, czy użytkownik uznawany jest za usuniętego, czy nie. Pole to występuje również w tabeli **ACCOUNTS**. Zastosowanie tego pola

motywuje występowaniem referencji do kont, a co za tym idzie pośrednio do użytkowników w tabeli transakcje. Jest to konieczne, ponieważ ze względu na swoją specyfikę, transakcja nie może zostać usunięta.

Cały schemat bazy opiera się na tym, że zachowuje ona informacje o wykonanych transakcjach. Gdybym miał opisać tę bazę jednym zdaniem, to powiedziałbym, że „zapisuje kto, kiedy, komu, ile wysłał”.

Architektura programu

Backend

Jak już wspomniałem we wstępie, najważniejszym kryterium podczas projektowania aplikacji była modułowość. Myślę, że udało mi się osiągnąć zamierzony cel. Użyłem następującej kombinacji:

- obiekt połączenia z bazą
- repozytorium bazowe(generyczne)
- modele encji
- modele DTO
- repozytoria szczegółowe(dla poszczególnych encji/modeli)
- serwisy(dla poszczególnych funkcjonalności)

Modele to odpowiedniki encji z bazy danych zmapowane do obiektów w Javie. Są wiernym odzwierciedleniem naszych rekordów z bazy. Wygenerowałem je przy użyciu pluginu JPA Buddy, a następnie trochę je dopracowałem(między innymi zmieniłem generowanie id na manualne).

Modele DTO to specjalne obiekty, których używa się do wykonywania operacji. Używałem ich głównie podczas modyfikowania oraz tworzenia danego rekordu. Stworzyłem także interfejs **ImappableTo<Model>**, który pozwala na zmapowanie DTO na odpowiedni pełnoprawny obiekt np. UserDTO -> User. Podczas mapowania zachodzi np. generowanie kluczy czy też hashowanie hasła. Zastosowanie obiektów DTO ułatwia także uniemożliwienie modyfikowania pól, które nie powinny być modyfikowane.

Obiekt połączenia z bazą- jest to klasa „statyczna” (w javie nie ma statycznych klas, chodzi mi tutaj o to, że udostępnia jedynie statyczną funkcjonalność), która tworzy obiekt SessionFactory raz w trakcie działania programu, tak aby inne komponenty mogły z niego korzystać. Jest to ważne ponieważ obiekt stworzenie obiektu SessionFactory to dość zasobochłonny proces. W moim programie jest ona nazwana **HibernateConnectUtility**.

Repozytorium bazowe(generyczne)- jest to klasa, która po zainicjalizowaniu przy użyciu odpowiednich typów pozwala na operacje na bazie danych z wykorzystaniem tychże typów. Umożliwia ona operacje bazowe CRUD- tworzenie, odczyt, modyfikację oraz usuwanie. Załóżmy, że tworzymy ją podając klasę User jako typ. Możemy wtedy za jej pomocą uzyskać, zmodyfikować, stworzyć oraz usunąć z bazy obiekty typu User. Klasa ta była kluczowa dla stworzenia programu, który posiada jak najmniej zduplikowanego kodu. Nazwałem ją **GenericRepository**.

Repozytoria szczegółowe są to klasy, które wykorzystując wewnątrz siebie instancję typowaną GenericRepository umożliwiają dostęp do encji konkretnego typu z bazy. Są to klasy będące tzw. wrapperami dla GenericRepository i umożliwiają dostęp do danych bardziej skrojony na miarę danego typu. Konwencja nazewnictwa, którą przyjąłem to ModelRepository (np. **UserRepository**, **AccountRepository**).

Serwisy to natomiast klasy skupione na konkretnej funkcjonalności. Zawierają one w sobie instancje odpowiednich repozytoriów, dzięki którym mogą wykonać swoje operacje. Są nastawione na konkretne działanie i ich przydatność jest jasno określona, nie są uniwersalne. Konwencja nazewnictwa dla serwisów

to FunkcjonalnośćService (np. **AuthorizationService**, który udostępnia funkcjonalność logowania). Kolejnym ważnym elementem w architekturze mojej aplikacji jest mechanizm cache. Zainstalowałem go poprzez użycie centralnej klasy udostępniającej swoje statyczne pola, jako odwzorowanie aktualnego stanu aplikacji. Zawiera ona informacje o aktualnie zalogowanym użytkowniku, spis jego kont oraz transakcji. Jest aktualizowana podczas ładowania danej strony, a także poprzez wywołanie niektórych funkcjonalności udostępnionych przez serwisy. Np. pozbywa się danych o użytkowniku, kiedy wywołujemy funkcję `authorizationService.SignOut()`. Klasę nazwałem **CurrentUserAppState**.

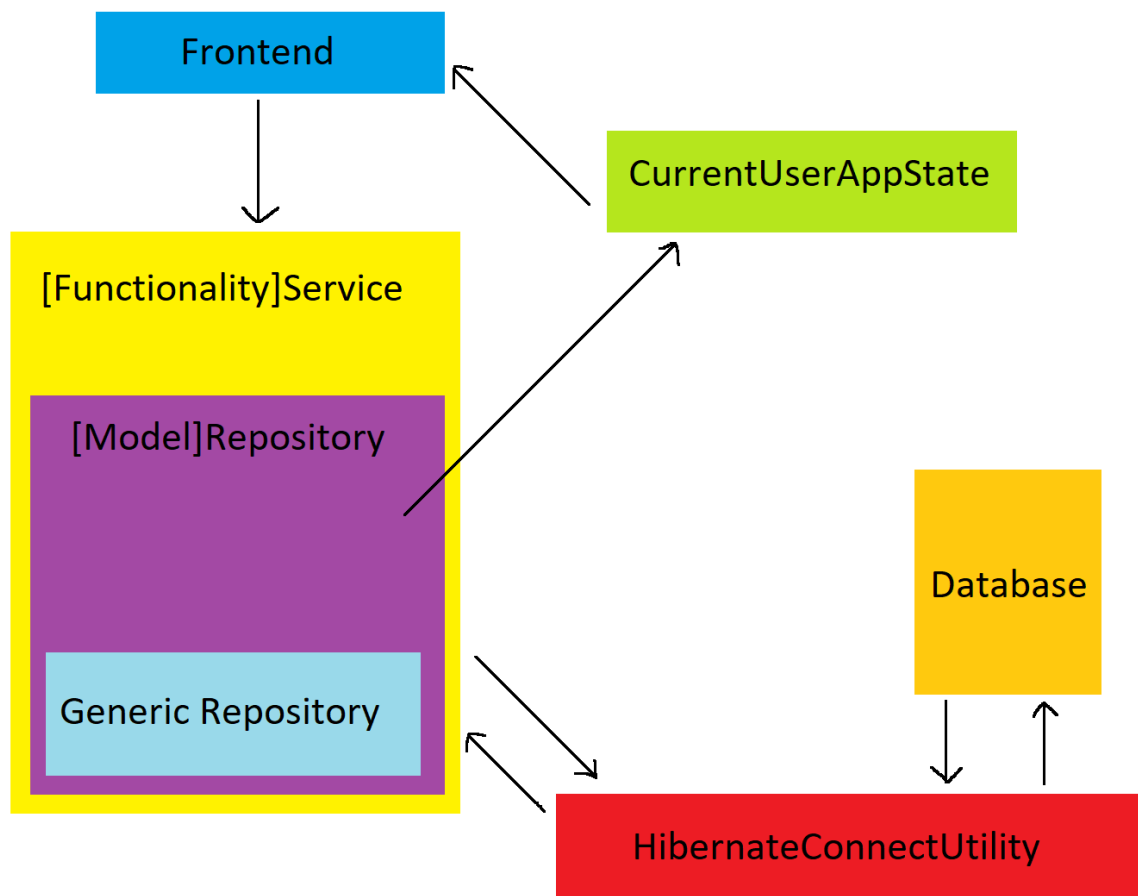


Diagram ukazujący architekturę programu.

Warto dodać, że każdego z tych komponentów można używać osobno. Np. do logowania możemy użyć `SignInService` i będzie to najwygodniejsze, ale możemy zrobić to też na niższym poziomie abstrakcji przy użyciu `UserRepository` lub `GenericRepository<User>` lub zrobić to bezpośrednio przez `HibernateConnectUtility` wywołując po prostu odpowiednie query.

Frontend

Interfejs zbudowany jest na podstawie odpowiednich scen zapisanych jako FXML. Nazywam je widokami. Ich nazwa w projekcie to `[NazwaWidoku]Screen`. Do każdego widoku przypisany jest kontroler a jego nazwa to `[NazwaWidoku]Controller`. Odpowiednie widoki budowałem przy użyciu programu Scene Builder. Jest to dość wygodne narzędzie. Głównym założeniem w przypadku budowy strony wizualnej projektu było to, żeby dane były pobierane tylko i wyłącznie z obiektu `CurrentUserAppState`, który służy

jako cache. Pozwala to zredukować ilość zapytań jakie wysyłamy do bazy oraz zapewnia bezpieczeństwo, ponieważ ładowane są tylko dane aktualnie zalogowanego użytkownika. Jeśli natomiast chcemy z poziomu frontu wykonać jakąś akcję, to wtedy używamy wtedy odpowiedniego serwisu.

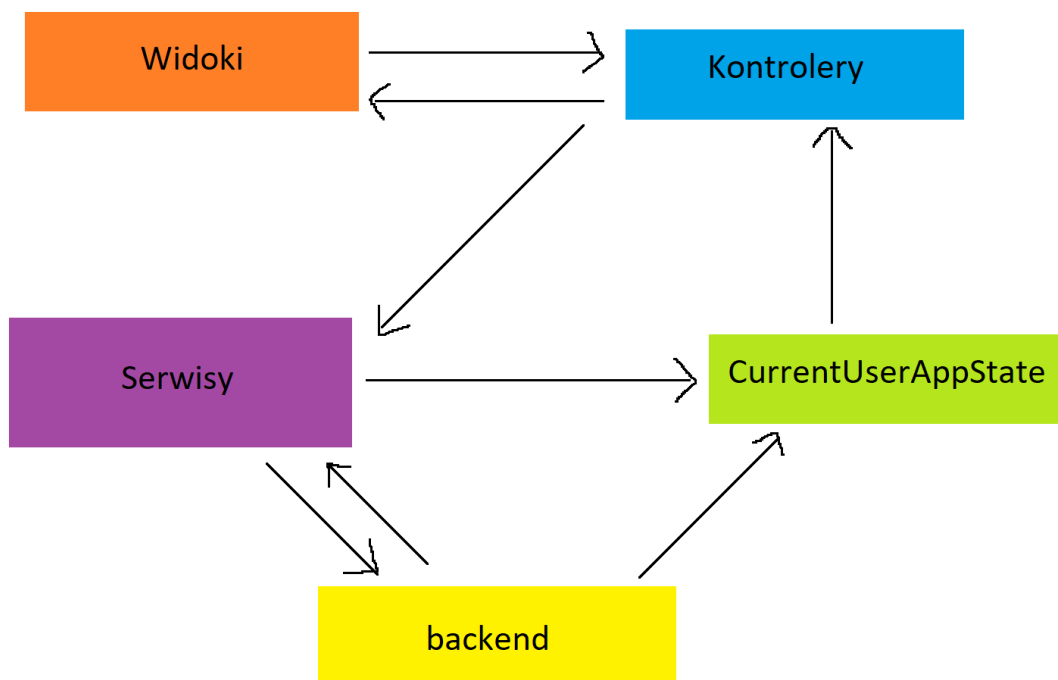
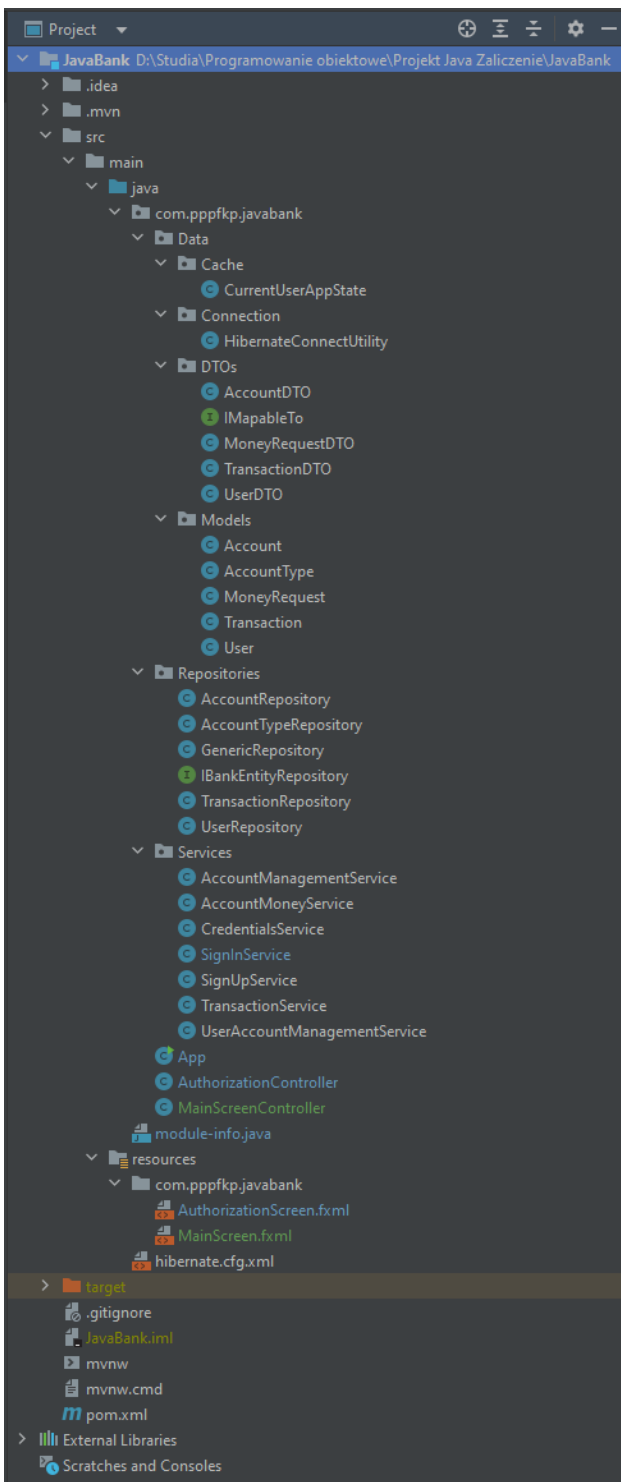


Diagram ukazujący strukturę program.

Logiczne uporządkowanie plików w projekcie



Główny folder dzieli się na podfoldery: Data, Repositories, Services. Luźno wrzucone do głównego folderu są kontrolery, ponieważ Scene builder nie pozwala na wygodną pracę z kontrolerem, który nie jest w tym samym folderze co widok. Gdyby istniała taka opcja, to stworzyłbym folder controllers i tam umieścił odpowiednie klasy. Bezpośrednio w root folderze programu jest także klasa startowa programu- App.

W folderze Data znajdują się foldery: Cache, Connection, DTOs, Models.

Myślę, że uporządkowanie plików oraz konwencje nazewnicze, które przyjąłem czynią ten projekt całkiem czytelny.

Przepływ danych w programie

Przepływ danych przedstawię na przykładzie encji account. Operacje będą bardzo podobne w przypadku obcowania z innymi encjami. Operacja, którą posłużę się w przykładzie to wypłacanie pieniędzy z konta.

1. Musimy znaleźć odpowiedni serwis, który będzie posiadał pomocną metodę. W tym przypadku jest to AccountMoneyService.
2. Lokalizujemy odpowiednią metodę. Nazywa się ona WithdrawMoneyFromAccount oraz przyjmuje następujące argumenty- ammountToWithdraw (ilość pieniędzy, którą chcemy wypłacić) typu BigDecimal oraz accountId (numer konta) typu String.
3. Wywołanie funkcji jest bardzo proste. Założmy, że chcemy wypłacić 1000zł z konta o numerze 81200112344175603185079988. Wystarczy stworzyć instancję klasy AccountMoneyService, w konstruktorze podać instancję AccountRepository, a następnie podczas wywołania funkcji podać te argumenty.

```
AccountMoneyService accountMoneyService = new AccountMoneyService(new
AccountRepository(HibernateConnectUtility.getSessionFactory()));

accountMoneyService.WithdrawMoneyFromAccount(new BigDecimal(1000), "81200112344175603185079988");
```

4. Funkcja zwróci nam ilość pieniędzy, która pozostanie na koncie po wykonaniu operacji.

Jeśli chodzi o ten najwyższy poziom abstrakcji, to jako konsument serwisu mamy wystarczające informacje, aby korzystać z tej funkcji. Aby lepiej zrozumieć jego działanie możemy jednak zajrzeć głębiej. Zaczniemy od działania metody w serwisie.

```
public BigDecimal WithdrawMoneyFromAccount(BigDecimal ammountToWithdraw, String accountId) {
    Account account = accountRepository.GetAccountById(accountId);
    if (account == null) {
        errors.add("Podane konto nie istnieje!");
        return null;
    }
    if (ammountToWithdraw.compareTo(new BigDecimal(0)) < 0) {
        errors.add("Kwota musi być większa od zera!");
        return null;
    }
    BigDecimal currentAmount = account.getBalance();
    if (ammountToWithdraw.compareTo(currentAmount) > 0) {
        errors.add("Nie posiadasz wystarczających środków na koncie!");
        return null;
    }
    AccountDTO dto = new AccountDTO(account);
    BigDecimal newBalance = currentAmount.subtract(ammountToWithdraw);
    dto.setBalance(newBalance);
    errors = dto.ValidateAll();
    if (!errors.isEmpty()) {
        return null;
    }
    if (accountRepository.UpdateAccount(dto, accountId)) {
        return newBalance;
    } else {
        errors.add("Błąd serwera!");
        return null;
    }
}
```

Każdy serwis posiada listę błędów, tak aby podczas interakcji z nim możliwe było ich odczytywanie np. z poziomu kontrolera widoku.

1. Funkcja pobiera z repozytorium konto o podanym numerze i sprawdza czy istnieje.
2. Funkcja sprawdza zasadność operacji- czy mamy wystarczające środki oraz czy nie próbujemy wypłacić ujemnej kwoty.
3. Funkcja tworzy dto na podstawie pobranego konta (poprzez konstruktor przyjmujący instancję konta jako argument).
4. Funkcja modyfikuje stan konta w dto konta.
5. Funkcja validuje dto używając do tego metody opkreslonej przez interfejs ImapableTo, który musi rozszerzać każde dto używane przez repozytorium.
6. Jeśli validacja przeszła pomyślnie to wykonuje się metoda UpdateAccount() z repozytorium.
7. Jeśli wynik jest pozytywny (zwrócona wartość logiczna true) to funkcja zwraca nowy stan konta.

Przyjrzyjmy się idąc dalej klasie AccountRepository. Jest ona wewnętrznym prywatnym polem serwisu. Jest ona bardzo prosta, ponieważ dzięki zastosowaniu generycznego repozytorium głównie zajmuje się routowanie do odpowiednich funkcji.

1. Podczas wywołania metody GetAccountById() funkcja po prostu wywołuje metodę GetSingleRecordById() udostępnianą przez repozytorium bazowe.

```
public Account GetAccountById(String id) {  
    return baseRepository.GetSingleRecordById(id);  
}
```

2. Wykonanie metody update. Tutaj sytuacja jest podobna.

```
public boolean UpdateAccount(AccountDTO dto, String oldAccountId) {  
    return baseRepository.UpdateRecord(dto, oldAccountId);  
}
```

Idąc dalej, zajrzyjmy do GenericRepository- repozytorium bazowe, które w AccountRepository miało postać pola prywatnego baseRepository. Tworząc instancję GenericRepository należy ją otypować następującymi typami- GenericRepository<TypEncji, TypDTO(musi implementować ImapableTo<TypEncji>), TypId>. W przypadku encji Account wygląda to tak:

```
private GenericRepository<Account, AccountDTO, String> baseRepository;
```

1. Metoda GetSingleRecord jest bardzo prosta. Pobiera id danego typu, następnie tworzy nową sesję oraz wywołuje metodę session.get(). Następnie zamyka sesję i zwraca rekord.

```
DataType GetSingleRecordById(IdType id) {  
    Session session = sessionFactory.openSession();  
    DataType record = session.get(dataTypeClass, id);  
    session.close();  
    return record;  
}
```

2. W przypadku metody `UpdateRecord()` sytuacja jest już trochę bardziej skomplikowana. Najpierw tworzymy listę błędów, którą wypełniamy wartościami z metody `dto.ValidateUpdatable()` (dlatego właśnie nasze `dto` musiało implementować interfejs `ImapableTo<TypEncji>`). Następnie otwieramy sesję. Jeśli nie ma błędów to przechodzimy dalej. Następnie rozpoczynamy transakcję. Pobieramy konto o danym `id`, następnie sprawdzamy czy nie jest `null`. Następnie mapujemy w naszym obiekcie wartości modyfikowalne używając obiektu `dto`.

```
boolean UpdateRecord(DataDTOType dto, IdType id) {
    List<String> validationErrors = dto.ValidateUpdatable();
    Session session = sessionFactory.openSession();

    if (!validationErrors.isEmpty()) {
        session.close();
        return false;
    }

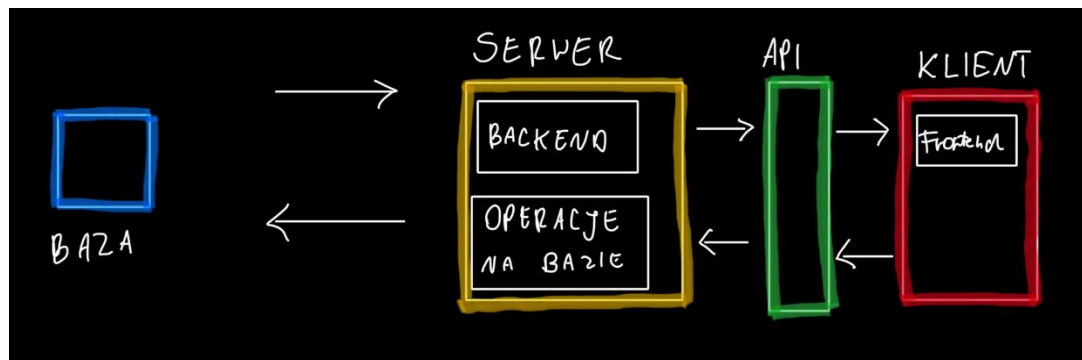
    try {
        session.beginTransaction();
        DataType data = session.get(dataTypeClass, id);
        if (data == null) {
            session.getTransaction().rollback();
            session.close();
            return false;
        }
        dto.MapToEntityTypeUpdateRecord(data);
        session.getTransaction().commit();
    } finally {
        session.close();
    }

    return true;
}
```

Obiekt encji `Account` to zwyczajny POJO. Zawiera pola, które chcemy odtworzyć w Javie z naszej bazy. Zawiera także adnotacje opisujące relacje i role. Jeśli chodzi o `dto`, to jest to obiekt, który może być mapowany z lub na encję właściwą. Posiada także metody do validacji, jednak nie implementowałem ich, bo u mnie walidacja odbywa się na poziomie serwisu.

Wady programu i błędy architektury

Według mnie największą wadą programu jest fakt, że próbuje jednocześnie być serwerem, jak i klientem. Prawidłowo zaprojektowana aplikacja tego typu powinna wyglądać inaczej. Wszystkie transakcje, logika operacji oraz walidacja powinny być wykonywane po stronie serwera, natomiast serwer powinien udostępniać odpowiednie API, z którego mogą skorzystać developerzy aplikacji klienckich. Umożliwiło by to ukrycie bazy danych przed światem zewnętrznym oraz sprawiłoby, że klient będzie mógł korzystać jedynie z określonych przez programistę funkcjonalności.



Po stronie klienta należy także wykonywać walidację, ponieważ nie obciążamy wtedy serwera, a użytkownik ma wtedy szybszy feedback.

Rozwiązanie, które zastosowałem sprawiło też problem z bezpieczeństwem danych. Chodzi mi o connection stringa. Jest to ciąg znaków, który zawiera nazwę bazy, adres serwera, login i hasło dostępu do bazy. Musiałem zapisać go w programie, tak aby miał on dostęp do bazy. Jest to niebezpieczne rozwiązanie, ale z uwagi na to, że udostępniam ten program jedynie prowadzącemu zajęcia, mogłem je dopuścić. Problem ten nie występuje w przypadku, gdy używamy poprawnej architektury, ponieważ postronne osoby nie mają dostępu do serwera, a jedynie do API udostępnionego przez serwer. Dzięki temu połączenie jest bezpiecznie wykonane, a jego detale są schowane przed światem zewnętrznym.

Kolejną wadą jest brak testów. Zaznaczę, że wykonywałem testy manualne, ale brakuje jednak testów automatycznych. Powinny być wykonane testy jednostkowe oraz integracyjne. Wykonując testy można stosować tzw. mock dependencies. Polega to na tym, że np. w przypadku testu funkcjonalności dodawania użytkowników podpinamy jako bazę danych bazę, która nie posiada użytkowników, których chcemy dodać.

Zalety programu

Myślę, że jak największą zaletę programu wymieniałbym łatwość dodawania nowych funkcjonalności. Jeśli chcemy poszerzyć program, to wystarczy napisać odpowiedni serwis, który będzie spełniał określone zadanie. Chciałem uzyskać taki stan, w którym dodanie nowej rzeczy nie psuje tych już istniejących.

Kolejna zaleta to przystosowanie do użycia Dependency Injection. Budowałem komponenty w taki sposób, aby było możliwe zastosowanie DI. Działa to na tej zasadzie, że specyfikujemy jaką zależność ma wstrzyknąć program, jeśli w konstruktorze będziemy mieli dany typ.

Podczas tworzenia aplikacji poprawiłem swoje umiejętności programistyczne, ponieważ musiałem rozwiązać wiele problemów. Dla mnie wielką zaletą tego programu jest fakt, że potrafiłem zobaczyć i zastosować tutaj wzorce projektowe takie jak Dependency Inversion, Singleton oraz ~MVC .

Kilka ciekawych rozwiązań zastosowanych w projekcie

Rozwiązania w moim projekcie, które uważam za najciekawsze to:

1. Hashowanie haseł.
2. Generowanie numerów kont.

Hashowanie haseł

W pierwszej wersji programu hasła przechowywane były w bazie danych jako niezaszyfrowane ciągi znaków. Jeśli użytkownik posiadał hasło „Password”, to w bazie danych w rekordzie tego użytkownika w kolumnie Password typu VARCHAR istniało po prostu pole z zawartością „Password”. Było to podejście bardzo naiwne, a wręcz głupie. Podczas tworzenia drugiej wersji projektu dowiedziałem się jak wygląda proces enkrypcji hasła i zastosowałem w tym celu bibliotekę **bcrypt**.

Jak działa enkrypcja haseł?

O procesie enkrypcji haseł można przeczytać np. tu [Salt \(cryptography\) - Wikipedia](#).

Zastosowałem technikę hashowania z saltem. W skrócie chodzi o to, że podczas tworzenia hasła, do tekstu wpisanego przez użytkownika dodawany jest losowy ciąg znaków, czyli tzw. salt. Następnie ciąg złożony z tekstu użytkownika(plaintext) oraz salta jest poddawany hashowaniu. Istnieją do tego specjalne funkcje. W bazie danych zapisywany jest utworzony hash oraz salt. Podczas autentykacji użytkownika proces hashowania się powtarza i powstałe hashe są do siebie porównywane. Jeśli są identyczne, to hasło jest prawidłowe. Salt potrzebny jest po to, aby wprowadzić „losowość” do algorytmu. Dzięki temu, że salt jest losowy, to nawet użytkownicy, którzy mają takie same hasła, będą mieli inne hashe.

W praktyce musiałem użyć trzech funkcji udostępnionych przez bibliotekę bcrypt:

-gensalt()

-hashpwd()

```
String passwordHash = BCrypt.hashpw(password, BCrypt.gensalt( log_rounds: 12));
```

-checkpwd()

```
BCrypt.checkpw(password, user.getPasswordHash());
```

Generowanie numerów kont

Pierwsza wersja mojego programu modelowała numer konta jako zwykłą liczbę całkowitą generowaną przez bazę jako id. Nie było to zbyt wierne odwzorowanie rzeczywistości dlatego postanowiłem to zmienić. Rozwiązanie jakie zastosowałem to przyjęcie arbitralnego numeru banku oraz oddziału. Uznałem, że mój bank ma jeden oddział, bo jest to bank internetowy.

Kody jakie przyjąłem to:

-kod banku: 2001

-kod oddziału: 1234

Budowa numeru konta

Numer konta ma 26 cyfr.

Dwie pierwsze cyfry to tzw. suma kontrolna. Jest ona obliczana na podstawie ostatnich 16 cyfr. Do jej obliczenia użyłem następującego algorytmu:

1. Weź 16 cyfr z końca numeru.
2. Zsumuj je ze sobą.
3. Przeprowadź operację modulo 99.

Następne 4 cyfry to kod banku.

Kolejne 4 cyfry to kod oddziału.

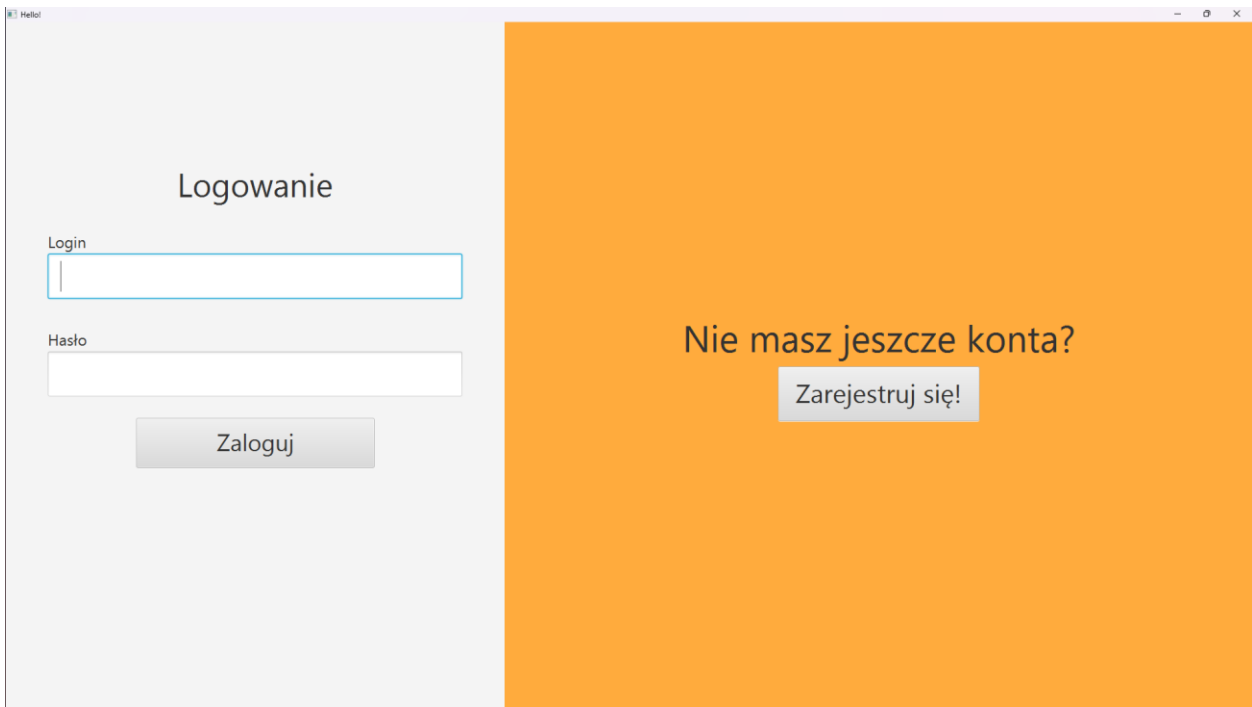
Ostatnie 16 cyfr to losowe cyfry. Jest to liczba na tyle duże, że praktycznie nie ma co się martwić o duplikaty.

Funkcja której użyłem do generowania numerów kont:

```
private String generateAccountNumber() {  
    Random rand = new Random();  
  
    long x = (long)(rand.nextDouble()*10000000000000000L);  
  
    String s = String.valueOf(sumOfDigits(x)) + "20011234" + String.format("%016d", x);  
    System.out.println(s);  
    return s;  
}
```

```
private int sumOfDigits(long number) {  
    long sum = 0;  
    while (number > 0) {  
        long lastDigit = number % 10;  
        sum += lastDigit;  
        number /= 10;  
    }  
    return Math.toIntExact(sum % 99);  
}
```


Użycie programu



Logowanie

Login

Hasło

Zaloguj

Nie masz jeszcze konta?

Zarejestruj się!

Ekran powitalny

Mamy tutaj możliwość rejestracji, bądź zalogowania. Domyślnie stworzony jest użytkownik testUser o haśle Password.



Wróć

Login

Hasło

Numer telefonu

Imię

Nazwisko

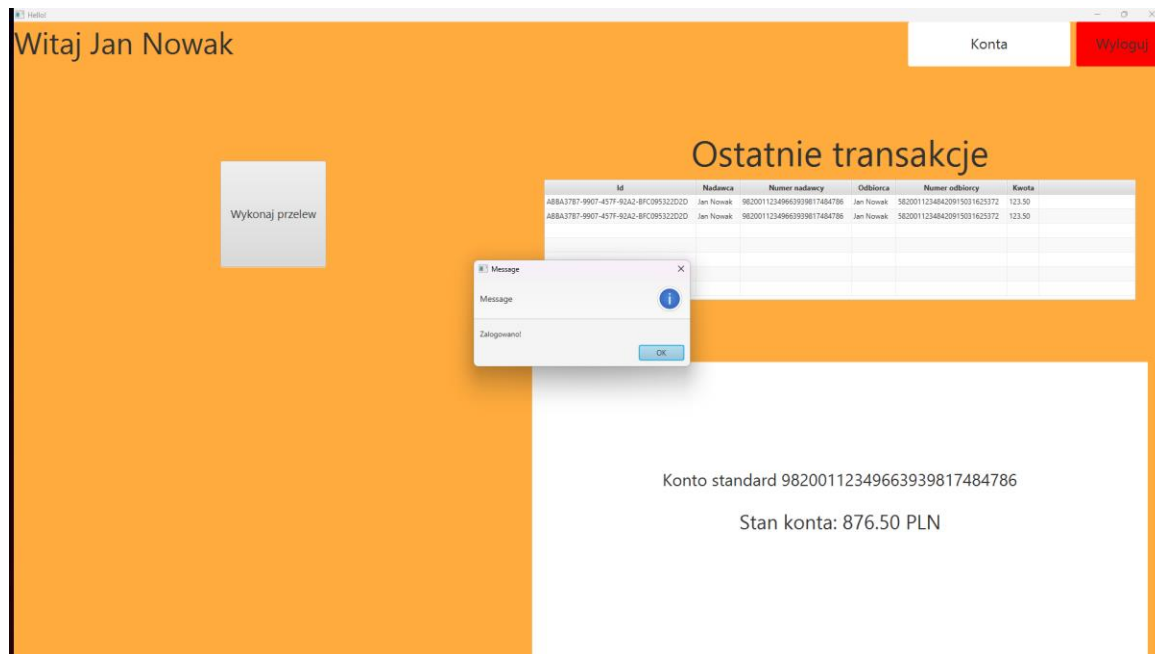
PESEL

Email

Data urodzenia

Zarejestruj

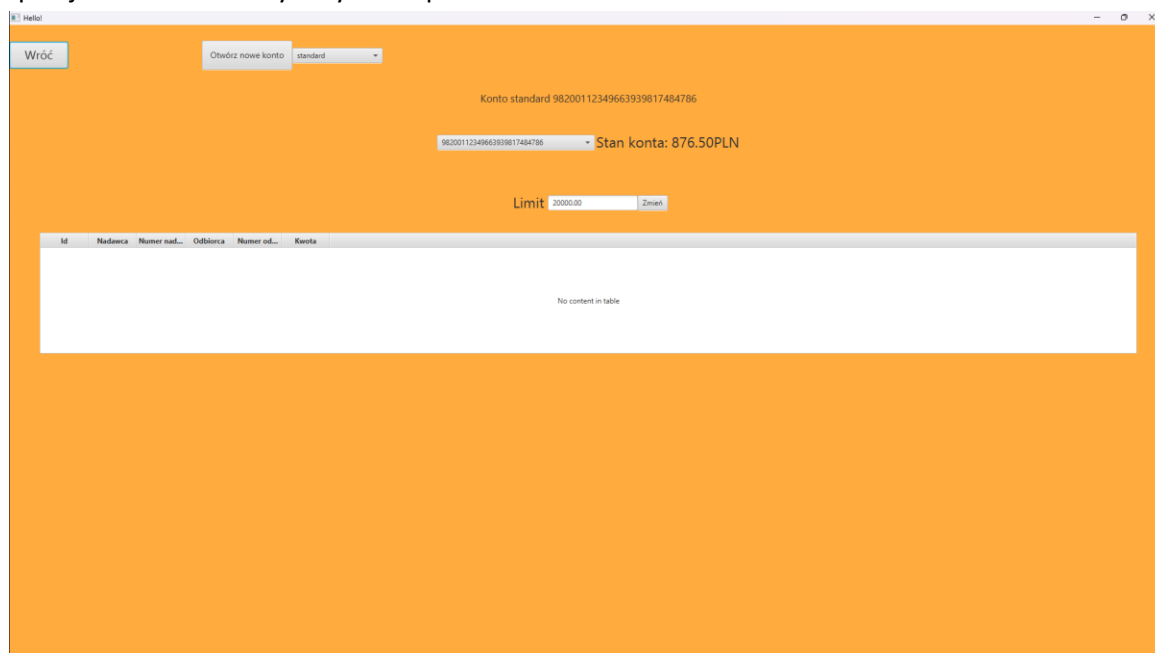
Gdy klikniemy przycisk zarejestruj ukaże się nam ekran rejestracji. Możemy tu stworzyć nowego użytkownika.



Po pomyślnej autoryzacji przejdziemy do ekranu głównego.

Możemy stąd podjąć następujące akcje:

- wylogowanie się
- przejsięcie do szczegółów kont
- przejsięcie do ekranu wykonywania przelewów



Z poziomu widoku kont możemy edytować konto(ustawić nowy limit), zobaczyć transakcję dla wybranego konta, czy też otworzyć nowe konto wybranego typu.

Wróć

Wykonaj przelew

Z: 9820011234963090817484786

do:

kwota: PLN

tytuł:

Wykonaj przelew

Z poziomu widoku wykonywania transakcji możemy wykonać przelew na wybrany przez nas numer konta.

Podsumowanie

Projekt ten był dla mnie wyzwaniem. Sądzę, że tworząc go nabyłem wiele przydatnych umiejętności oraz miałem możliwość zastosowania i utrwalenia wiedzy zdobytej podczas tego semestru na przedmiocie Programowanie Obiektowe. W procesie tworzenia projektu zastosowałem wiedzę z zakresu programowania obiektowego, programowania w języku Java, tworzenia interfejsów w JavaFX oraz obsługiwaną połączenia z bazą danych.