Финальное задание: MiniBank

Описание проекта

В этом задании вы создадите консольное банковское приложение на Java с использованием **Spring Core**. Приложение позволит создавать пользователей и банковские счета, пополнять и снимать деньги, переводить средства между счетами и просматривать информацию о пользователях.

Все данные хранятся **в памяти** (в коллекциях), без базы данных. Никакого Spring Boot и автоконфигураций — только Spring Core, ручная настройка контекста и управление зависимостями через аннотации.

Это задание поможет закрепить на практике ключевые концепции Spring: **IoC-контейнер**, **внедрение зависимостей**, **конфигурацию через аннотации** и работу с **PropertySource**.

Необходимые знания

Spring Core

- Создание и конфигурация Spring-контекста
- Аннотации `@Component`, `@Configuration`, `@PropertySource`
- Внедрение зависимостей через `@Autowired`
- Использование `@Value` для извлечения параметров из `application.properties`
- Понимание жизненного цикла бинов (`@PostConstruct`)

Java Core

- Классы, коллекции (`List`, `Map`), исключения
- Чтение данных из консоли (`Scanner`)

Детальное описание задания

Как работает программа

Приложение запускается и ожидает ввода команд от пользователя через консоль. Пользователь вводит одну из доступных команд, программа выполняет действие, печатает результат и снова предлагает ввести команду. Программа работает в цикле до ввода команды `EXIT`.

Доступные команды

- **USER_CREATE** — создание нового пользователя
- **SHOW_ALL_USERS** — отображение списка всех пользователей
- **ACCOUNT_CREATE** — создание нового счёта для пользователя
- **ACCOUNT_DEPOSIT** — пополнение счёта
- **ACCOUNT_WITHDRAW** — снятие средств со счёта
- **ACCOUNT_TRANSFER** — перевод средств между счетами
- **ACCOUNT_CLOSE** — закрытие счёта
- **EXIT** — завершение работы программы

Описание каждой команды

USER_CREATE Запрашивает логин нового пользователя. Проверяет, что пользователя с таким логином ещё нет в системе. Создаёт пользователя с автоматически назначенным ID и одним счётом с начальным балансом из настроек.

SHOW_ALL_USERS Не требует входных данных. Выводит список всех пользователей с информацией об их счетах.

ACCOUNT_CREATE Запрашивает ID пользователя. Создаёт новый счёт с уникальным ID и дефолтным балансом из настроек.

ACCOUNT_DEPOSIT Запрашивает ID счёта и сумму. Пополняет счёт на указанную сумму.

ACCOUNT_WITHDRAW Запрашивает ID счёта и сумму. Снимает указанную сумму со счёта. Если средств недостаточно — выводит ошибку.

ACCOUNT_TRANSFER Запрашивает ID счёта отправителя, ID счёта получателя и сумму перевода. При переводе между счетами **разных** пользователей взимается комиссия — получатель получает сумму за вычетом комиссии. Между собственными счетами одного пользователя перевод без комиссии.

ACCOUNT_CLOSE Запрашивает ID счёта. Закрывает счёт, переводя остаток средств на первый оставшийся счёт пользователя. Если у пользователя только один счёт — закрыть его нельзя.

EXIT Завершает работу программы.

Настройки из application.properties

```
account.default-amount=500
account.transfer-commission=0.02
```

- **account.default-amount** — начальный баланс каждого нового счёта
- **account.transfer-commission** — комиссия за перевод между счетами разных пользователей (доля от суммы перевода). Отправитель теряет полную сумму, получатель получает `сумма * (1 - комиссия)`

Модель данных: POJO-классы

> **Что такое POJO и зачем это понимать?**
>
> *POJO (Plain Old Java Object) — это обычный Java-объект, который не зависит ни от какого фреймворка. У него есть поля, конструктор, геттеры и сеттеры — и больше ничего.*
>
> *`User` и `Account` в этом проекте — именно такие объекты. Они **не являются Spring-бинами** и **не должны** быть помечены аннотацией `@Component`. Их создают в коде через `new` каждый раз, когда пользователь вводит команду на создание пользователя или счёта.*
>
> *Типичная ошибка начинающих — пометить `User` или `Account` как `@Component`. Не делайте так. Spring-бин существует, как правило, в единственном экземпляре и управляется контейнером. А пользователей и счетов может быть сколько угодно, и создаются они динамически по запросу.*

User — пользователь системы

- `id` (int) — уникальный идентификатор
- `login` (String) — логин пользователя
- `accountList` (List<Account>) — список счетов

Account — банковский счёт

- `id` (int) — уникальный идентификатор счёта
- `userId` (int) — ID пользователя-владельца
- `moneyAmount` (int) — текущий баланс

Что должно быть Spring-компонентом

Следующие классы управляются Spring-контейнером и должны быть бинами:

- **ApplicationConfiguration** (`@Configuration`) — конфигурация контекста с `@ComponentScan` и `@PropertySource`
- **AccountProperties** (`@Component`) — хранит настройки из `application.properties`, внедрённые через `@Value`
- **UserService** (`@Component`) — сервис управления пользователями: создание, поиск по ID, получение списка
- **AccountService** (`@Component`) — сервис управления счетами: создание, пополнение, снятие, перевод, закрытие
- **ConsoleInput** (`@Component`) — единая точка чтения и валидации ввода
- **OperationsConsoleListener** (`@Component`) — слушает консольный ввод и вызывает нужные методы сервисов

Эти компоненты внедряются друг в друга через конструктор. Например, `OperationsConsoleListener` получает `UserService` и `AccountService` через `@Autowired`.

Обработка ошибок

Программа должна корректно обрабатывать ошибки и выводить понятные сообщения пользователю.

Примеры ситуаций, которые нужно обработать

- Создание пользователя с уже существующим логином
- Обращение к несуществующему пользователю или счёту
- Ввод отрицательной суммы для пополнения, снятия или перевода
- Попытка снять больше, чем есть на счёте

- Попытка закрыть единственный счёт пользователя

Пример сообщения:

```
Error: insufficient funds on account id=1, moneyAmount=0, attempted
withdraw=100
```

Рекомендуемая структура проекта

```
src/
 └── main/
      ├── java/
      |     ├── config/
      |     |     └── ApplicationConfiguration.java
      |     ├── user/
      |     |     ├── User.java
      |     |     └── UserService.java
      |     ├── account/
      |     |     ├── Account.java
      |     |     ├── AccountService.java
      |     |     └── AccountProperties.java
      |     ├── console/
      |     |     ├── ConsoleInput.java
      |     |     └── OperationsConsoleListener.java
      |     ├── operations/
      |     |     ├── ConsoleOperationType.java
      |     |     ├── OperationCommand.java
      |     |     └── commands/
      |     └── Main.java
      └── resources/
            └── application.properties
```

Пакет `operations` хранит типы команд и их реализации, а пакет `console` отвечает за ввод данных и основной цикл обработки.

Рекомендуемый порядок реализации

Если не знаете с чего начать, попробуйте двигаться в таком порядке:

1. Создайте `ApplicationConfiguration` с аннотациями `@Configuration`, `@ComponentScan` и `@PropertySource`
2. Создайте модели данных (`User`, `Account`) — обычные POJO с полями, конструктором, геттерами и сеттерами
3. Реализуйте `AccountProperties` — компонент, который через `@Value` получает настройки из файла
4. Реализуйте `UserService` и `AccountService`. Данные храните в `Map<Integer, User>` и `Map<Integer, Account>`
5. Реализуйте `ConsoleInput` и `OperationsConsoleListener` — используйте `Scanner` для чтения команд из консоли
6. Создайте точку входа `Main.java` — создание `AnnotationConfigApplicationContext` и запуск `OperationsConsoleListener`
7. Добавьте обработку ошибок с понятными сообщениями

Зависимости (pom.xml)

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.1.5</version>
    </dependency>

    <dependency>
        <groupId>jakarta.annotation</groupId>
        <artifactId>jakarta.annotation-api</artifactId>
        <version>2.1.1</version>
    </dependency>
</dependencies>
```

Версии зависимостей могут быть более новыми.

Пример работы программы

Значения настроек для этого примера: `account.default-amount=500`, `account.transfer-commission=0.02`

```
MiniBank started. Type EXIT to stop.
Available commands: USER_CREATE, SHOW_ALL_USERS, ACCOUNT_CREATE,
ACCOUNT_DEPOSIT, ACCOUNT_WITHDRAW, ACCOUNT_TRANSFER, ACCOUNT_CLOSE, EXIT

Enter command:
USER_CREATE
Enter login:
Pavel Sorokin
User created: User{id=1, login='Pavel Sorokin', accounts=[Account{id=1,
userId=1, moneyAmount=500}]}

Enter command:
USER_CREATE
Enter login:
Vasya Pypkin
User created: User{id=2, login='Vasya Pypkin', accounts=[Account{id=2,
userId=2, moneyAmount=500}]}

Enter command:
ACCOUNT_CREATE
Enter user id:
1
Account created: Account{id=3, userId=1, moneyAmount=500}

Enter command:
ACCOUNT_DEPOSIT
Enter account id:
3
Enter amount:
100
Deposited 100 to account 3. New balance: 600

Enter command:
ACCOUNT_TRANSFER
Enter source account id:
```

```
3
Enter target account id:
1
Enter amount:
600
Transfer completed from account 3 to account 1. Amount: 600 (no commission,
same user)

Enter command:
SHOW_ALL_USERS
User{id=1, login='Pavel Sorokin', accounts=[Account{id=1, userId=1,
moneyAmount=1100}, Account{id=3, userId=1, moneyAmount=0}]}
User{id=2, login='Vasya Pypkin', accounts=[Account{id=2, userId=2,
moneyAmount=500}]}

Enter command:
ACCOUNT_TRANSFER
Enter source account id:
1
Enter target account id:
2
Enter amount:
1100
Transfer completed from account 1 to account 2. Amount: 1100, commission: 22,
recipient received: 1078

Enter command:
SHOW_ALL_USERS
User{id=1, login='Pavel Sorokin', accounts=[Account{id=1, userId=1,
moneyAmount=0}, Account{id=3, userId=1, moneyAmount=0}]}
User{id=2, login='Vasya Pypkin', accounts=[Account{id=2, userId=2,
moneyAmount=1578}]}

Enter command:
ACCOUNT_WITHDRAW
Enter account id:
1
Enter amount:
100
Error: insufficient funds on account id=1, moneyAmount=0, attempted
withdraw=100
```

```
Enter command:
ACCOUNT_CLOSE
Enter account id to close:
3
Account 3 closed. Remaining balance 0 transferred to account 1.

Enter command:
SHOW_ALL_USERS
User{id=1, login='Pavel Sorokin', accounts=[Account{id=1, userId=1,
moneyAmount=0}]}
User{id=2, login='Vasya Pypkin', accounts=[Account{id=2, userId=2,
moneyAmount=1578}]}

Enter command:
EXIT
MiniBank stopped.
```

Обратите внимание на расчёт комиссии в примере:

- Перевод 600 между своими счетами (account 3 → account 1) — комиссии нет, получатель получает всю сумму
- Перевод 1100 другому пользователю (account 1 → account 2) — комиссия 2%: отправитель теряет 1100, получатель получает `1100 * (1 - 0.02) = 1078`. Итого на счёте получателя: `500 + 1078 = 1578`
- При закрытии счёта 3 (баланс 0) остаток переносится на первый счёт пользователя

Как проверить свою работу

Проверка ввода/вывода

Запустите приложение и попробуйте каждую из доступных команд. Убедитесь, что программа корректно принимает команды и выводит результаты. Проверьте реакцию на некорректные данные: отрицательные числа, несуществующие ID, пустой ввод.

Проверка логики операций

- **Создание пользователя и счёта** — пользователь создаётся, у него появляется счёт с дефолтным балансом
- **Пополнение** — сумма корректно зачисляется на нужный счёт
- **Снятие** — успешное снятие и ошибка при недостаточном балансе
- **Перевод между своими счетами** — без комиссии, полная сумма
- **Перевод другому пользователю** — с комиссией, проверьте что суммы сходятся
- **Закрытие счёта** — остаток переносится, единственный счёт закрыть нельзя

Проверка настроек

Измените значения `account.default-amount` и `account.transfer-commission` в `application.properties`. Перезапустите приложение и убедитесь, что изменения применяются: другой начальный баланс, другой процент комиссии.

Проверка обработки ошибок

Попробуйте вызвать ошибки: создать пользователя с существующим логином, обратиться к несуществующему счёту, снять сумму больше баланса. Программа должна выводить понятные сообщения, а не падать с исключением.

Хорошее тестирование включает проверку не только успешных сценариев, но и граничных условий.

Подсказки

Если возникли трудности с какой-то частью задания, загляните в файл с подсказками.