8

Gestion des transactions

Au sommaire de ce chapitre

- Problèmes associés à la gestion des transactions
- Choisir une implémentation de gestionnaire de transactions
- Gérer les transactions par programmation avec l'API du gestionnaire de transactions
- Gérer les transactions par programmation avec un template de transaction
- Gérer les transactions par déclaration avec Spring AOP classique
- Gérer les transactions par déclaration avec des greffons transactionnels
- Gérer les transactions par déclaration avec l'annotation *@Transactional*
- Fixer l'attribut transactionnel de propagation
- Fixer l'attribut transactionnel d'isolation
- Fixer l'attribut transactionnel d'annulation
- Fixer les attributs transactionnels de temporisation et de lecture seule
- Gérer les transactions avec le tissage au chargement
- En résumé

Ce chapitre présente les concepts de base des transactions et les possibilités de Spring dans ce domaine. La gestion des transactions est un élément essentiel dans les applications d'entreprise car elle permet de garantir l'intégrité et la cohérence des données. Spring, en tant que framework d'applications d'entreprise, définit une couche abstraite au-dessus des différentes API de gestion des transactions. En tant que développeurs d'applications, nous pouvons nous servir des outils de Spring pour la gestion des transactions sans vraiment connaître ces API.

À l'instar des approches BMT (*Bean-Managed Transaction*) et CMT (*Container-Managed Transaction*) dans les EJB, la gestion des transactions dans Spring peut se faire par programmation ou par déclaration. Les fonctions de prise en charge des transactions dans Spring offrent une alternative aux transactions des EJB en apportant des possibilités transactionnelles aux POJO.

Dans la *gestion des transactions par programmation*, le code de gestion des transactions est incorporé dans les méthodes métier de manière à contrôler la validation (*commit*) et l'annulation (*rollback*) des transactions. En général, une transaction est validée lorsqu'une méthode se termine de manière normale, elle est annulée lorsqu'une méthode lance certains types d'exceptions. Avec cette gestion des transactions, nous pouvons définir nos propres règles de validation et d'annulation.

Toutefois, cette approche nous oblige à écrire du code de gestion supplémentaire pour chaque opération transactionnelle. Un code transactionnel standard est ainsi reproduit pour chacune des opérations. Par ailleurs, il nous est difficile d'activer et de désactiver la gestion des transactions pour différentes applications. Avec nos connaissances en POA, nous comprenons sans peine que la gestion des transactions est une forme de préoccupation transversale.

Dans la plupart des cas, la *gestion des transactions par déclaration* doit être préférée à la gestion par programmation. Le code de gestion des transactions est séparé des méthodes métier *via* des déclarations. En tant que préoccupation transversale, la gestion des transactions peut être modularisée par une approche POA. Le framework Spring AOP sert de fondation à la gestion déclarative des transactions dans Spring. Nous pouvons ainsi activer plus facilement les transactions dans nos applications et définir une politique transactionnelle cohérente. En revanche, cette forme de gestion est moins souple, car nous ne pouvons pas contrôler précisément les transactions à partir du code.

Grâce à un ensemble d'attributs transactionnels, nous pouvons configurer de manière très fine les transactions. Les attributs reconnus par Spring concernent la propagation, le niveau d'isolation, les règles d'annulation, les temporisations et le fonctionnement en lecture seule. Ils nous permettent ainsi d'adapter le comportement de nos transactions.

À la fin de ce chapitre, vous serez capable d'appliquer différentes stratégies de gestion des transactions dans vos applications. Par ailleurs, les attributs transactionnels vous seront devenus suffisamment familiers pour définir précisément des transactions.

8.1 Problèmes associés à la gestion des transactions

La gestion des transactions est un élément essentiel dans les applications d'entreprise car elle permet de garantir l'intégrité et la cohérence des données. Sans les transactions, les données et les ressources pourraient être endommagées et laissées dans un état incohérent. Cette gestion est extrêmement importante dans les environnements concurrents et répartis lorsque le traitement doit se poursuivre après des erreurs inattendues.

En première définition, une transaction est une suite d'actions qui forment une seule unité de travail. Ces actions doivent toutes réussir ou n'avoir aucun effet. Si toutes les actions réussissent, la transaction est validée de manière permanente. En revanche, si

l'une des actions se passe mal, la transaction est annulée de manière à revenir dans l'état initial, comme si rien ne s'était passé.

Le concept de transactions peut être décrit par les quatre propriétés ACID :

- **Atomicité.** Une transaction est une opération atomique constituée d'une suite d'opérations. L'atomicité d'une transaction garantit que toutes les actions sont entièrement exécutées ou qu'elles n'ont aucun effet.
- **Cohérence.** Dès lors que toutes les actions d'une transaction se sont exécutées, la transaction est validée. Les données et les ressources sont alors dans un état cohérent qui respecte les règles métier.
- **Isolation.** Puisque plusieurs transactions peuvent manipuler le même jeu de données au même moment, chaque transaction doit être isolée des autres afin d'éviter la corruption des données.
- **Durabilité.** Dès lors qu'une transaction est terminée, les résultats doivent survivre à toute panne du système. En général, les résultats d'une transaction sont écrits dans une zone de stockage persistant.

Pour comprendre l'importance de la gestion des transactions, prenons pour exemple l'achat de livres auprès d'une librairie en ligne. Tout d'abord, nous devons créer un nouveau schéma pour cette application dans notre base de données. Pour le moteur de base de données Apache Derby, la connexion se fait à l'aide des propriétés JDBC indiquées dans le Tableau 8.1.

Tableau 8.1 : Propriétés JDBC pour la connexion à la base de données de l'application

Propriété	*Valeur*
Classe du pilote	`org.apache.derby.jdbc.ClientDriver`
URL	`jdbc:derby://localhost:1527/bookshop;create=true`
Nom d'utilisateur	`app`
Mot de passe	`app`

Pour l'enregistrement des données de notre application de librairie, nous créons plusieurs tables.

```
CREATE TABLE BOOK (
    ISBN         VARCHAR(50)    NOT NULL,
    BOOK_NAME    VARCHAR(100)   NOT NULL,
    PRICE        INT,
    PRIMARY KEY (ISBN)
);
```

```
CREATE TABLE BOOK_STOCK (
    ISBN         VARCHAR(50)    NOT NULL,
    STOCK        INT            NOT NULL,
    PRIMARY KEY (ISBN),
    CHECK (STOCK >= 0)
);

CREATE TABLE ACCOUNT (
    USERNAME     VARCHAR(50)    NOT NULL,
    BALANCE      INT            NOT NULL,
    PRIMARY KEY (USERNAME),
    CHECK (BALANCE >= 0)
);
```

La table `BOOK` contient les informations de base sur un livre, comme son titre et son prix. L'ISBN du livre en est la clé primaire. La table `BOOK_STOCK` indique le stock de chaque livre. La quantité en stock est affectée d'une contrainte `CHECK` de manière à rester positive. Bien que la contrainte `CHECK` soit définie dans SQL-99, elle n'est pas reconnue par tous les moteurs de base de données. Si c'est le cas du vôtre, consultez sa documentation pour connaître la contrainte équivalente. Enfin, la table `ACCOUNT` enregistre les comptes des clients et leur solde, qui est garanti positif par une contrainte `CHECK`.

L'interface `BookShop` définit les opérations de notre librairie. Pour le moment, elle ne contient que l'opération `purchase()`.

```
package com.apress.springrecipes.bookshop;

public interface BookShop {

    public void purchase(String isbn, String username);
}
```

Puisque nous implémentons cette interface avec JDBC, nous créons la classe `JdbcBookShop` suivante. Pour mieux comprendre la nature des transactions, procédons sans l'aide de Spring JDBC.

```
package com.apress.springrecipes.bookshop;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

public class JdbcBookShop implements BookShop {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

```
    public void purchase(String isbn, String username) {
        Connection conn = null;
        try {
            conn = dataSource.getConnection();

            PreparedStatement stmt1 = conn.prepareStatement(
                    "SELECT PRICE FROM BOOK WHERE ISBN = ?");
            stmt1.setString(1, isbn);
            ResultSet rs = stmt1.executeQuery();
            rs.next();
            int price = rs.getInt("PRICE");
            stmt1.close();

            PreparedStatement stmt2 = conn.prepareStatement(
                    "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 " +
                    "WHERE ISBN = ?");
            stmt2.setString(1, isbn);
            stmt2.executeUpdate();
            stmt2.close();

            PreparedStatement stmt3 = conn.prepareStatement(
                    "UPDATE ACCOUNT SET BALANCE = BALANCE - ? " +
                    "WHERE USERNAME = ?");
            stmt3.setInt(1, price);
            stmt3.setString(2, username);
            stmt3.executeUpdate();
            stmt3.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {}
            }
        }
    }
}
```

Pour l'opération `purchase()`, nous exécutons trois requêtes SQL. La première obtient le prix du livre, tandis que la deuxième et la troisième mettent à jour le stock du livre et ajustent le solde du compte en conséquence.

Nous déclarons ensuite une instance de la librairie dans le conteneur Spring IoC de manière à offrir les services d'achat. Pour simplifier, nous utilisons un `DriverManagerDataSource` qui ouvre une nouvelle connexion à la base de données à chaque requête.

INFO

Pour accéder à une base de données du serveur Derby, vous devez inclure le fichier `derbyclient.jar` (situé dans le répertoire `lib` de l'installation de Derby) dans le chemin d'accès aux classes.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
            value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url"
            value="jdbc:derby://localhost:1527/bookshop;create=true" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>

    <bean id="bookShop" class="com.apress.springrecipes.bookshop.JdbcBookShop">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

Afin d'illustrer les problèmes qui peuvent survenir sans la gestion des transactions, supposons que la base de données de notre librairie contienne les données présentées dans les Tableaux 8.2 à 8.4.

Tableau 8.2 : Données d'exemple de la table `BOOK` pour le test des transactions

ISBN	*BOOK_NAME*	*PRICE*
0001	Le premier livre	30

Tableau 8.3 : Données d'exemple de la table `BOOK_STOCK` pour le test des transactions

ISBN	*STOCK*
0001	10

Tableau 8.4 : Données d'exemple de la table `ACCOUNT` pour le test des transactions

USERNAME	*BALANCE*
utilisateur1	20

Dans la classe `Main` suivante, le client `utilisateur1` achète le livre dont l'ISBN est `0001`. Puisque ce client ne dispose que de 20 € sur son compte, il ne peut pas acheter le livre.

```
package com.apress.springrecipes.bookshop;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
public class Main {

     public static void main(String[] args) {
          ApplicationContext context =
               new ClassPathXmlApplicationContext("beans.xml");

          BookShop bookShop = (BookShop) context.getBean("bookShop");
          bookShop.purchase("0001", "utilisateur1");
     }
}
```

Si nous exécutons cette application, nous recevons une exception `SQLException` car la contrainte `CHECK` sur la table `ACCOUNT` n'est pas respectée. Ce résultat était attendu, puisque nous avons tenté un débit supérieur au solde du compte. Toutefois, si nous examinons le stock de ce livre dans la table `BOOK_STOCK`, nous constatons qu'il a été diminué par cette opération ratée ! En effet, nous avons exécuté la deuxième requête SQL pour diminuer le stock avant de recevoir l'exception générée par la troisième.

L'absence d'une gestion des transactions a donc placé nos données dans un état incohérent. Pour éviter cette situation, nos trois requêtes SQL de l'opération `purchase()` doivent être exécutées au sein d'une même transaction. Si l'une des actions de la transaction échoue, l'intégralité de la transaction est annulée pour défaire les changements apportés par les actions exécutées.

Gérer les transactions avec la validation et l'annulation de JDBC

Lorsque la mise à jour d'une base de données se fait avec JDBC, chaque requête SQL est, par défaut, validée immédiatement après son exécution. Ce comportement se nomme *validation automatique*. Il ne nous permet pas de gérer des transactions dans nos opérations.

JDBC prend en charge une stratégie élémentaire de gestion des transactions qui consiste à invoquer explicitement les méthodes `commit()` et `rollback()` sur une connexion. Cependant, nous devons commencer par désactiver la validation automatique, qui est active par défaut.

```
package com.apress.springrecipes.bookshop;
...
public class JdbcBookShop implements BookShop {
     ...
     public void purchase(String isbn, String username) {
          Connection conn = null;
          try {
               conn = dataSource.getConnection();
               conn.setAutoCommit(false);
               ...
               conn.commit();
          } catch (SQLException e) {
               if (conn != null) {
```

```
                    try {
                        conn.rollback();
                    } catch (SQLException e1) {}
                }
                throw new RuntimeException(e);
            } finally {
                if (conn != null) {
                    try {
                        conn.close();
                    } catch (SQLException e) {}
                }
            }
        }
    }
```

La méthode `setAutoCommit()` permet de configurer le fonctionnement en validation automatique d'une connexion à une base de données. Par défaut, la validation automatique est active de manière à valider chaque requête SQL immédiatement après son exécution. Pour gérer les transactions, nous devons désactiver ce comportement par défaut et valider la connexion uniquement lorsque toutes les requêtes SQL ont été exécutées avec succès. Si l'une des requêtes échoue, nous devons annuler toutes les modifications apportées par cette connexion.

À présent, si nous exécutons à nouveau notre application, le stock du livre n'est pas affecté lorsque le solde d'un utilisateur est insuffisant pour acheter l'ouvrage.

Bien que nous puissions gérer les transactions en validant et en annulant explicitement les connexions JDBC, le code correspondant est un code standard que nous devons répéter dans différentes méthodes. Par ailleurs, ce code est propre à JDBC et doit être changé si nous décidons d'employer une autre technologie d'accès aux données. Pour simplifier la gestion des transactions, Spring propose donc un ensemble d'outils transactionnels indépendants de la technologie, en particulier des gestionnaires de transactions, un template de transaction et la gestion des transactions par déclaration.

8.2 Choisir une implémentation de gestionnaire de transactions

Problème

De manière générale, si notre application n'emploie qu'une seule source de données, nous pouvons simplement gérer les transactions en invoquant les méthodes `commit()` et `rollback()` sur la connexion à la base de données. En revanche, si les transactions concernent plusieurs sources de données ou si nous préférons utiliser les possibilités de gestion des transactions apportées par le serveur d'applications Java EE, nous pouvons opter pour JTA (*Java Transaction API*). Par ailleurs, nous risquons d'avoir à invoquer différentes API transactionnelles propriétaires en fonction des différents frameworks de correspondance objet-relationnel, comme Hibernate et JPA.