Spring Start Here

Chapter-13
Using transactions in Spring apps

Upcode Software
Engineer Team

-2024-

CONTENT

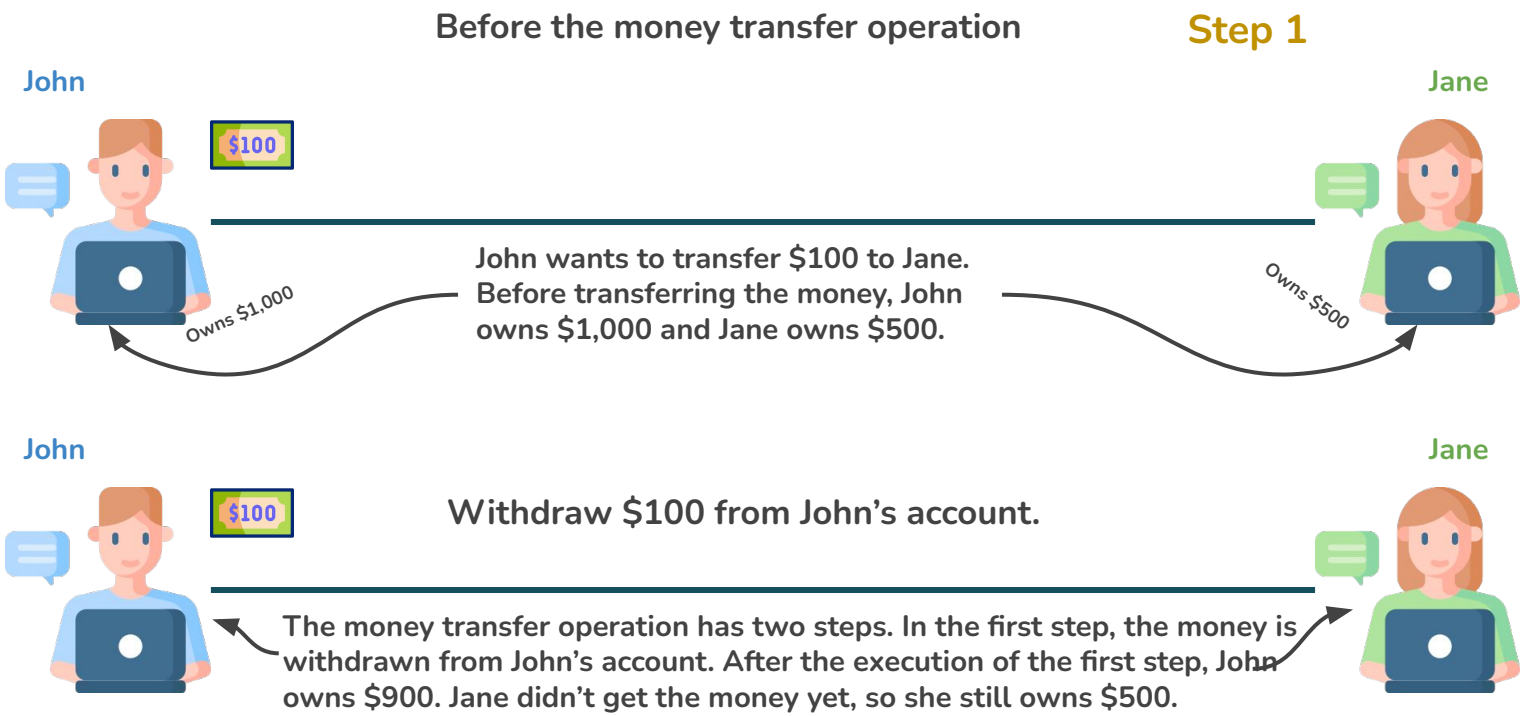
1. Introduction (1/5)

Transaction is a series of actions that fail as a group or complete entirely as a group, all these actions should be ***rollback*** in case of any failure, but if all of them complete then the transaction should be permanently ***committed***.

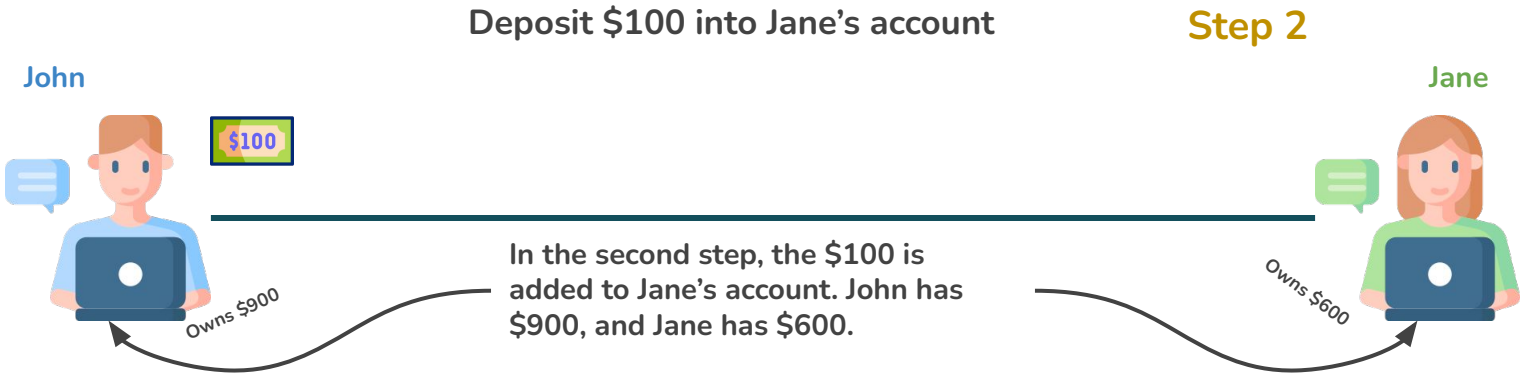
Let's we see one example on the next pages.
Suppose we implement an application used to share money - an electronic wallet.
In this application, a user has accounts where they store their money.
We implement a functionality to allow a user to transfer money from one account to another.
Considering a simplistic implementation for our example, this implies two steps (figure 1):

1. Withdraw money from the source account.
2. Deposit money into the destination account

1. Introduction (2/5)



1. Introduction (3/5)



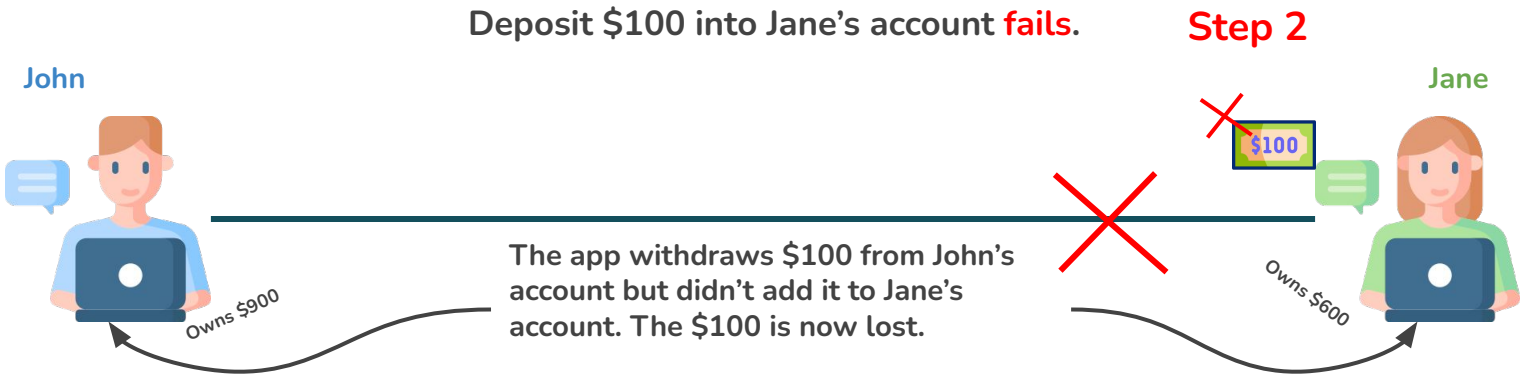
An example of a use case.
When transferring money from one account to another account, the app executes two operations:

- It subtracts the transferred money from the first account.
- Adds it to the second account.

We'll implement this use case, and we need to make sure its execution won't generate inconsistencies in data.

1. Introduction (4/5)

If the second step fails, we end up in a situation where the money has been taken from John's account, but Jane never got it. John will have $900 while Jane still has $500. Where did the $100 go? Figure 13.2 illustrates this behavior.



If one of the steps of a use case fails, data becomes inconsistent. For the money transfer example, if the operation that subtracts the money from the first accounts succeeds, but the operation that adds it to the destination account fails, money is lost.

1. Introduction (5/5)

To avoid such scenarios in which data becomes inconsistent, we need to make sure either both steps correctly execute or neither of them do. Transactions offer us the possibility to implement multiple operations that either correctly execute all or none.

Let's we discuss deeper about the **Transactions** with examples on the next pages...

2. Transactions (1/9)

A **transaction** is a defined set of mutable operations (operations that change data) that can either correctly execute them altogether ***or*** not at all. We refer to this as atomicity.

ACID Properties

Atomicity

Each transaction is treated as a single unit, which either succeeds completely or fails completely.

Consistency

Transactions only make changes to tables in predefined, predictable ways.

Isolation

Concurrent transactions don't interface with or affect one another.

Durability

Once a transaction has been committed, it will remain committed even in the case of a system failure.

2. Transactions (2/9)

ACID is an acronym that stands for atomicity, consistency, isolation, and durability. **ACID** properties ensure that a database transaction (a set of read, write, update, or delete operations) leaves the database in a consistent state even in the event of unexpected errors.

I believe that the new terms related to the Transaction should also be briefly discussed in the topic. Let's also briefly go over each of the types of atomicity related to ACID encountered here and how they are used on the next 4 pages.

2. Transactions (3/9)

Atomicity

- ***Definition***: Ensures that a transaction is an all-or-nothing operation. If any part of the transaction fails, the entire transaction is rolled back, leaving the database state unchanged.
- ***Spring Implementation***: Managed using the **@Transactional** annotation or programmatically with transaction managers. If an exception occurs, the transaction is automatically rolled back.

```
@Transactional
public void performTransaction() {
    // All operations within this method are part of a single transaction
}
```

2. Transactions (4/9)

Consistency

- ***Definition***: Ensures that a transaction brings the database from one valid state to another, maintaining all predefined rules, such as constraints, cascades, and triggers.
- ***Spring Implementation***: The developer must ensure business logic and database constraints are correctly defined. Spring helps by managing **transactions** and ensuring that operations conform to these constraints.

```
@Transactional
public void performConsistentTransaction() {
    // Ensure that business rules and constraints are followed
}
```

2. Transactions (5/9)

Isolation

- ***Definition***: Ensures that concurrently executed **transactions** do not interfere with each other. **Isolation** levels determine how transaction integrity is visible to other **transactions**.
- ***Spring Implementation***: Spring allows setting different **isolation** *levels* (e.g., READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE) using the **@Transactional** annotation.

```
@Transactional(isolation = Isolation.SERIALIZABLE)
public void performIsolatedTransaction() {
    // Operations are isolated according to the SERIALIZABLE level
}
```

2. Transactions (6/9)

Durability

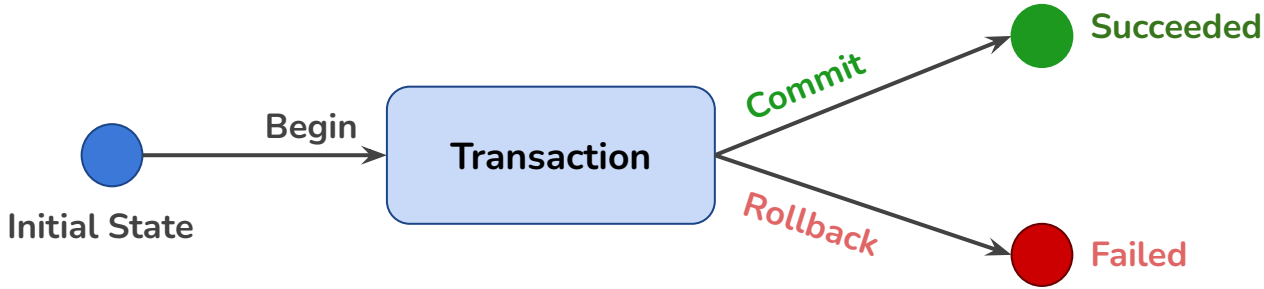
- ***Definition***: Ensures that once a **transaction** has been committed, the changes are permanent and will survive system failures.
- ***Spring Implementation***: Managed by the underlying database and its **transaction** log mechanisms. Spring ensures that once a **transaction** commits, the changes are reliably stored.

```
@Transactional
public void performDurableTransaction() {
    // Once committed, changes will persist even in case of system failure
}
```

2. Transactions (7/9)

Transactions are essential in apps because they ensure the data remains consistent if any step of the use case fails when the app already changed data.
Let's again consider a (simplified) transfer money functionality consisting of two steps:

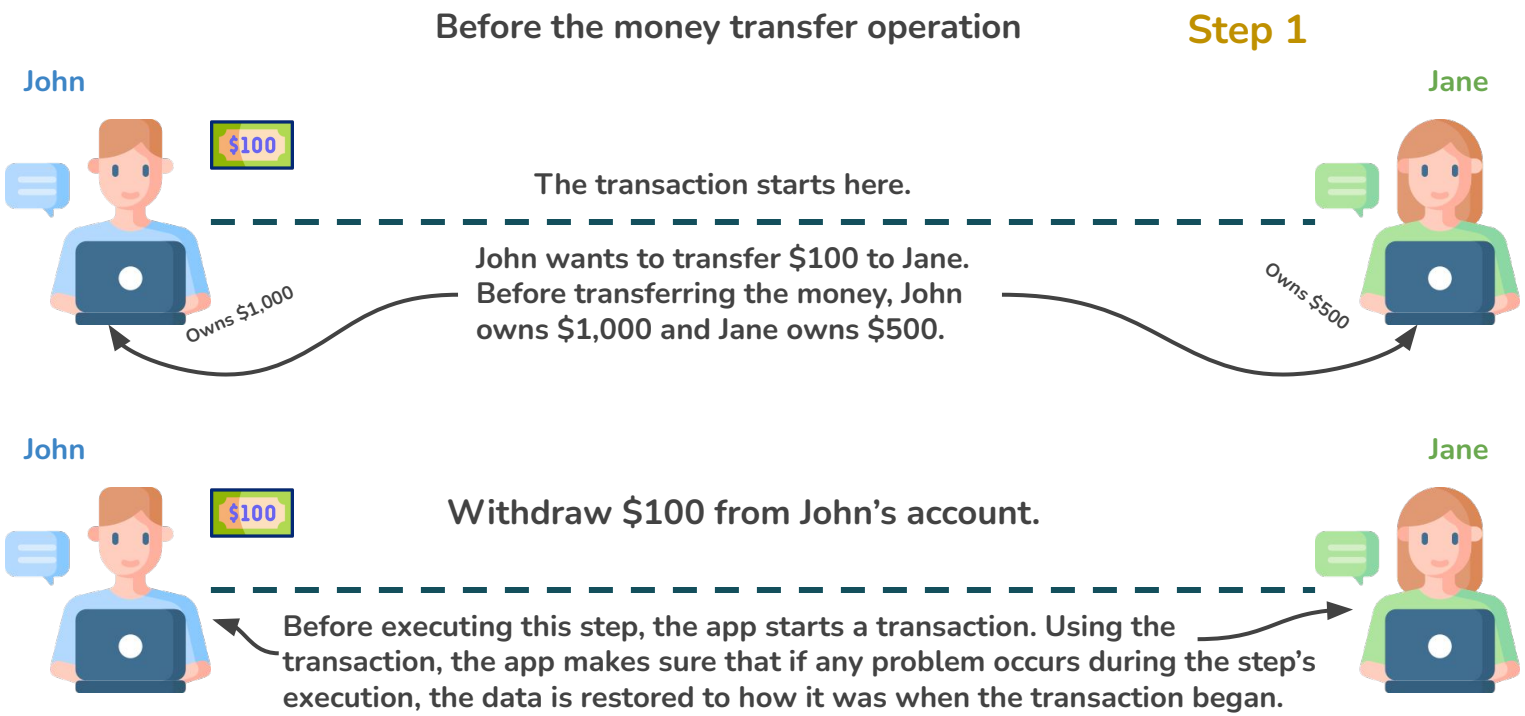
- ***Withdraw*** money from the source account.
- ***Deposit*** money into the destination account.



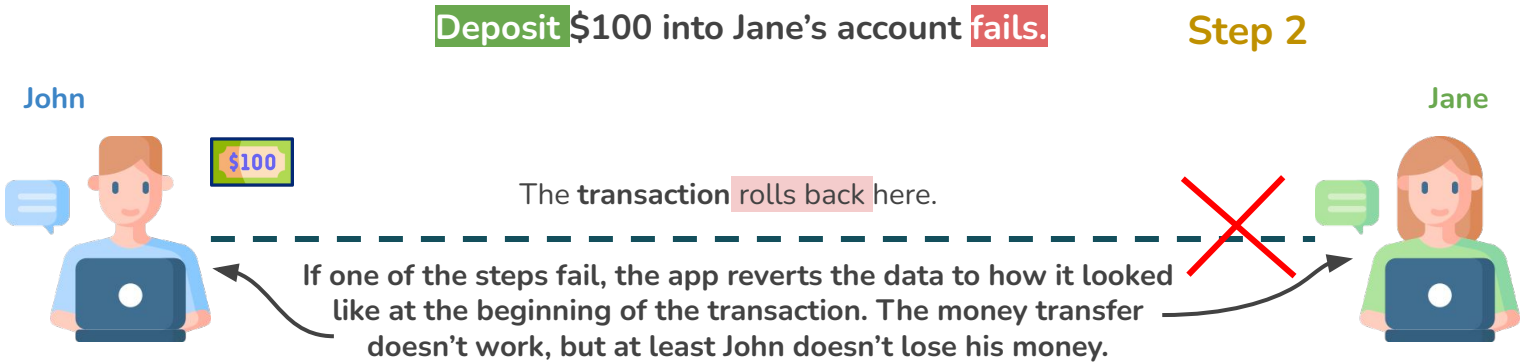
COMMIT The successful end of a transaction when the app stores all the changes made by the transaction's mutable operations.

ROLLBACK The transaction ends with rollback when the app restores the data to the way it looked at the beginning of the transaction to avoid data inconsistencies.

2. Transactions (8/9)



2. Transactions (9/9)

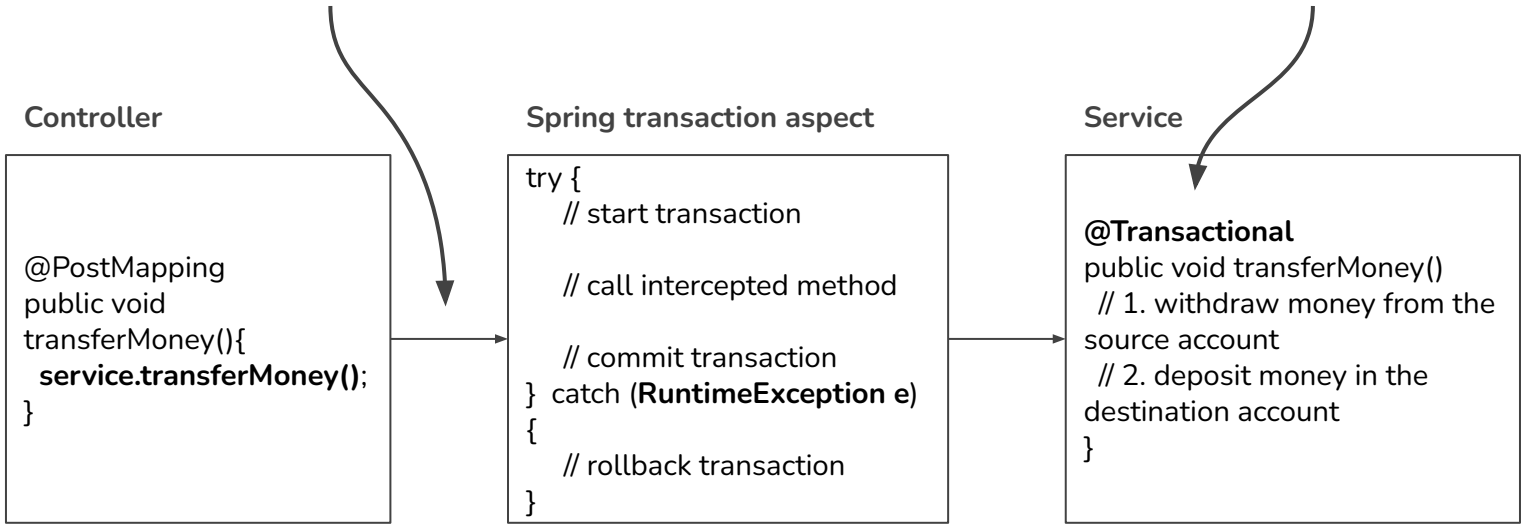


A transaction solves possible inconsistencies that could appear if any of the steps of a use case fail. With a transaction, if any of the steps fail, the data is reverted to how it was at the transaction start.

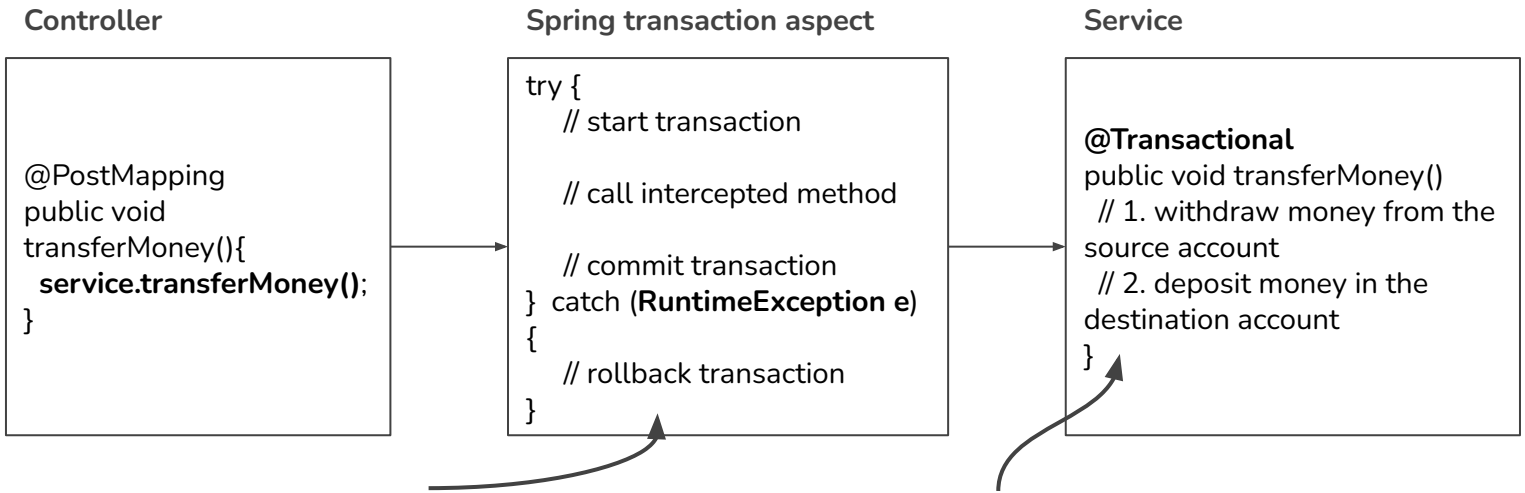
3. How transactions work in Spring (1/3)

Something (e.g., a controller action) calls the service method. Because the method is annotated with @Transactional, Spring configures an aspect that intercepts the call.

The @Transactional annotation is what tells the Spring transaction aspect to intercept this method.



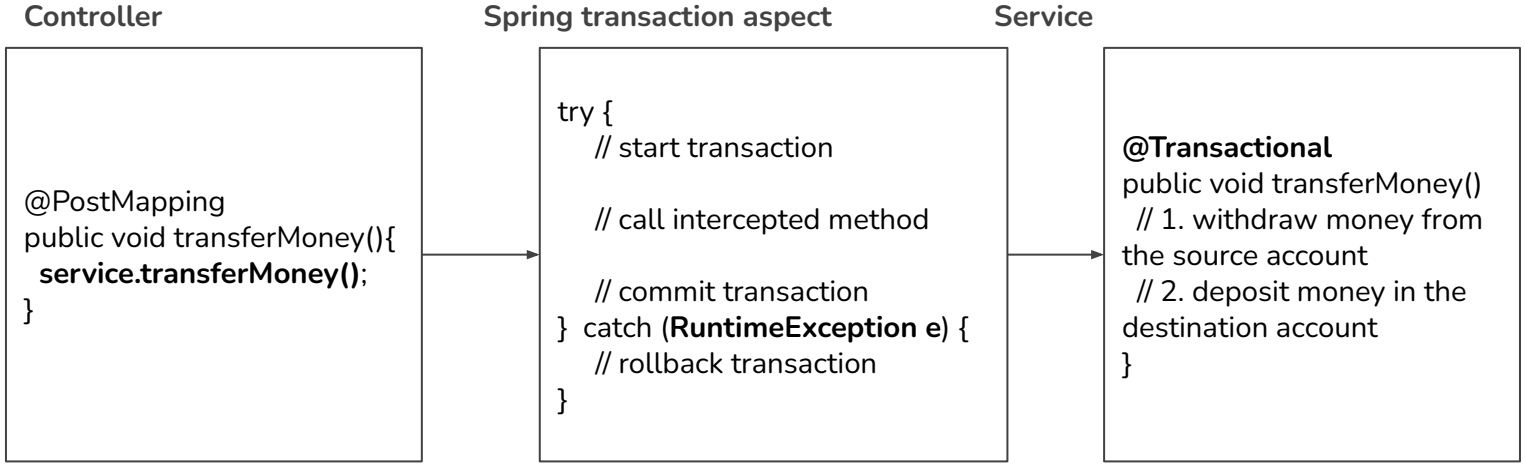
3. How transactions work in Spring (2/3)



This is a simplistic representation of the Spring transaction aspect logic. By default, if the intercepted method throws any runtime exception, the aspect rolls back the transaction. If the intercepted method didn't throw a runtime exception, the transaction is committed.

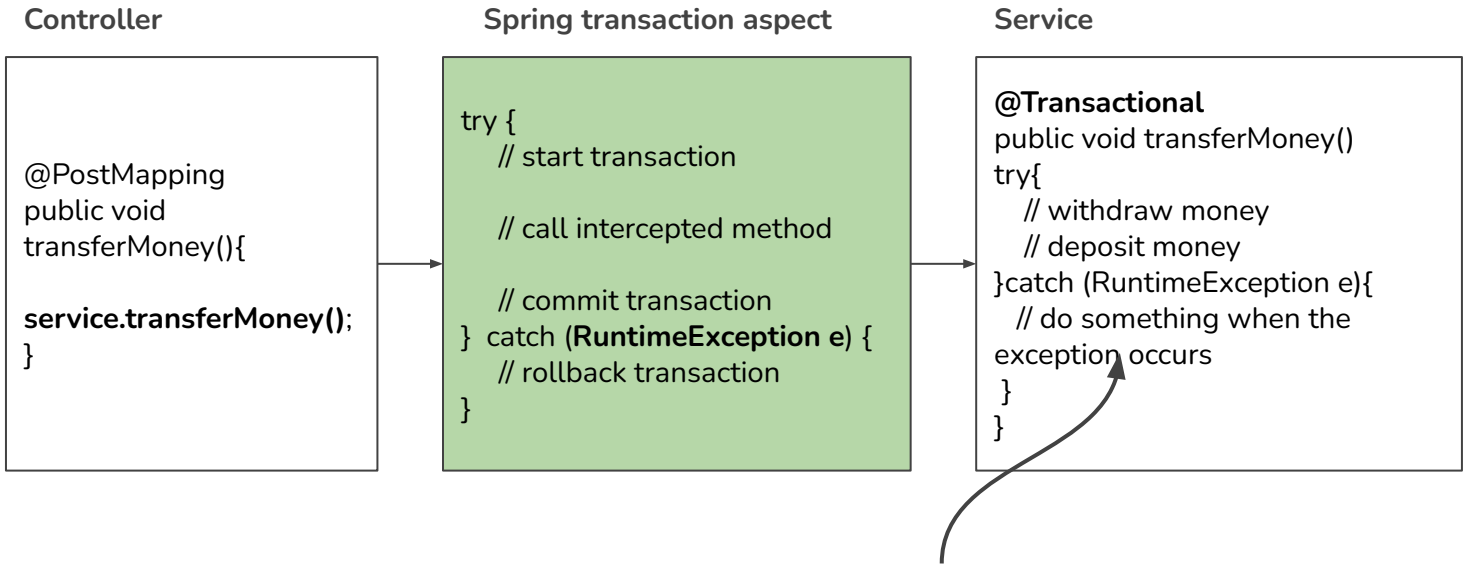
Because the whole method call is wrapped in the transaction, both steps are now in the transaction. If step 1 succeeds but step 2 throws a runtime exception, then the Spring transaction aspect will revert the changes made by step 1 with the transaction rollback operation.

3. How transactions work in Spring (3/3)

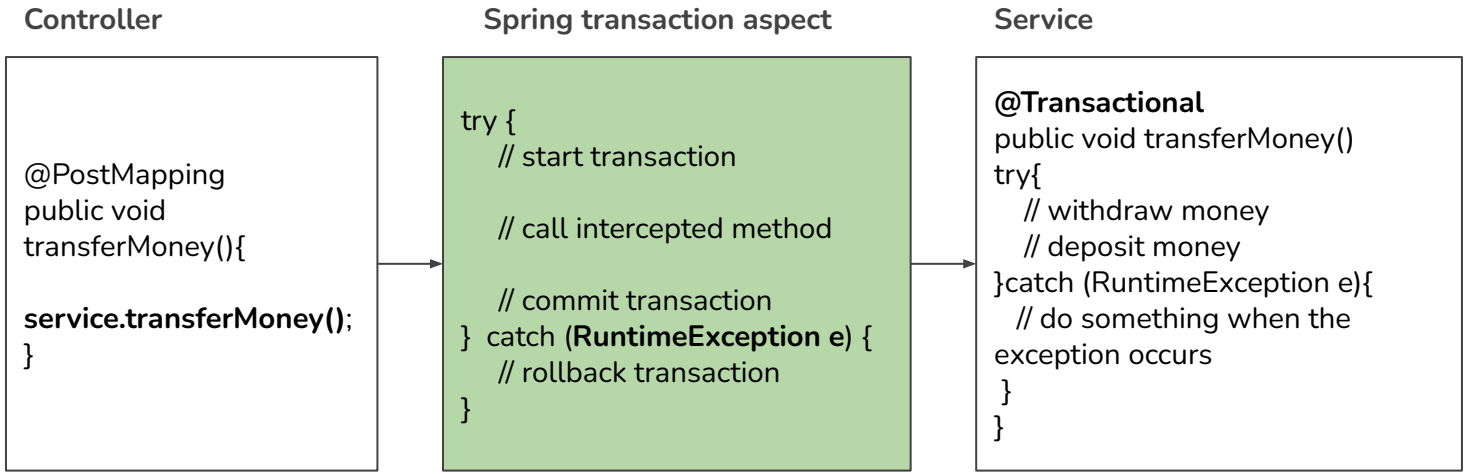


When we use the **@Transactional** annotation with a method, an aspect configured by Spring intercepts the method call and applies the transaction logic for that call. The app doesn't persist the changes the method makes if the method throws a runtime exception.

4. Using transactions in Spring apps(1/21)



4. Using transactions in Spring apps(2/21)

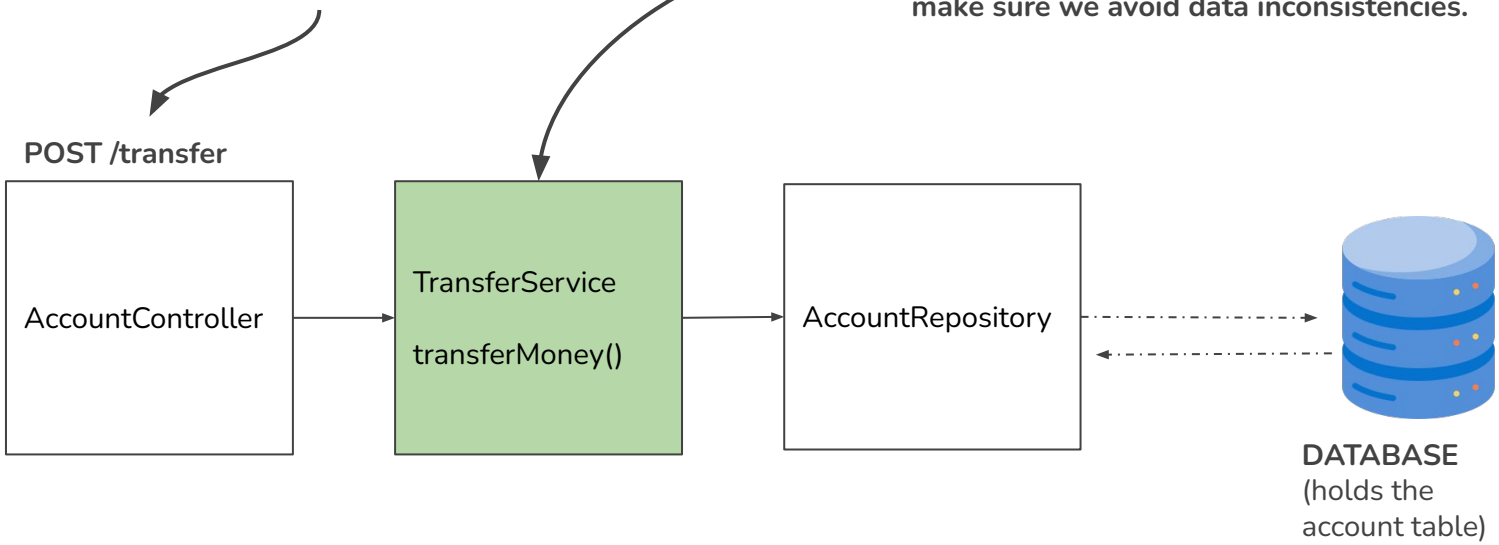


If a runtime exception is thrown inside the method, but the method treats the exception and doesn't throw it back to the caller, the aspect won't get this exception and will commit the transaction. When you treat an exception in a transactional method, such as in this case, you need to be aware the transaction won't be rolled back, as the aspect managing the transaction cannot see the exception.

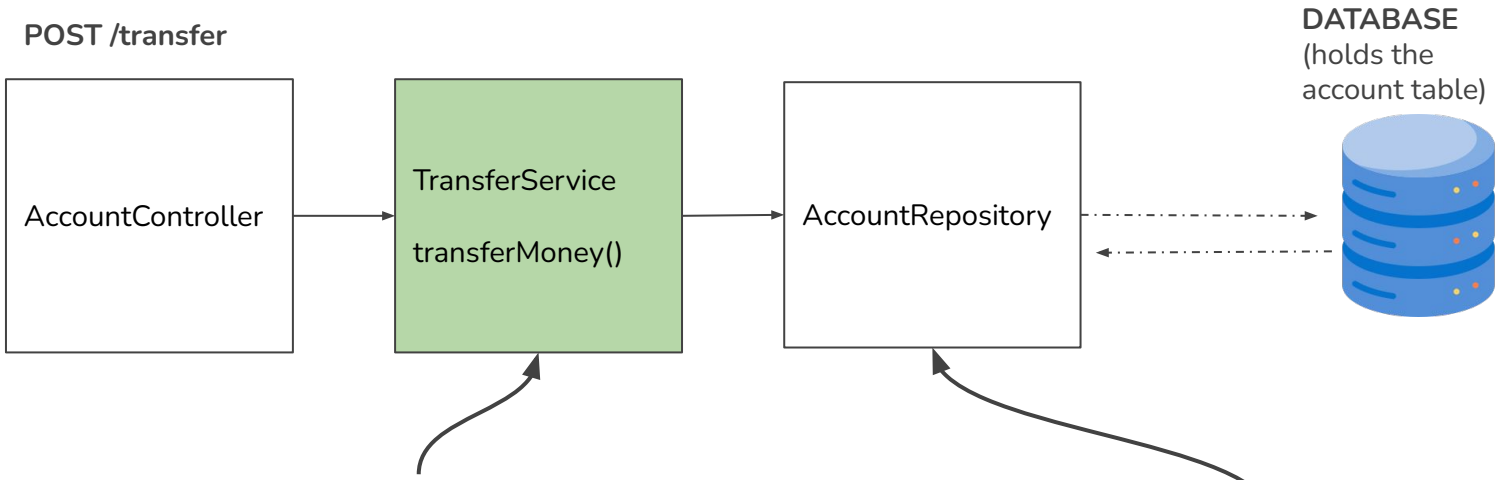
4. Using transactions in Spring apps(3/21)

The AccountController is a REST controller that exposes the POST /transfer endpoint. This endpoint offers a way to call the transfer money use case.

The TransferService implements the transfer money use case with the transferMoney() method. We need to execute this method in a transaction to make sure we avoid data inconsistencies.



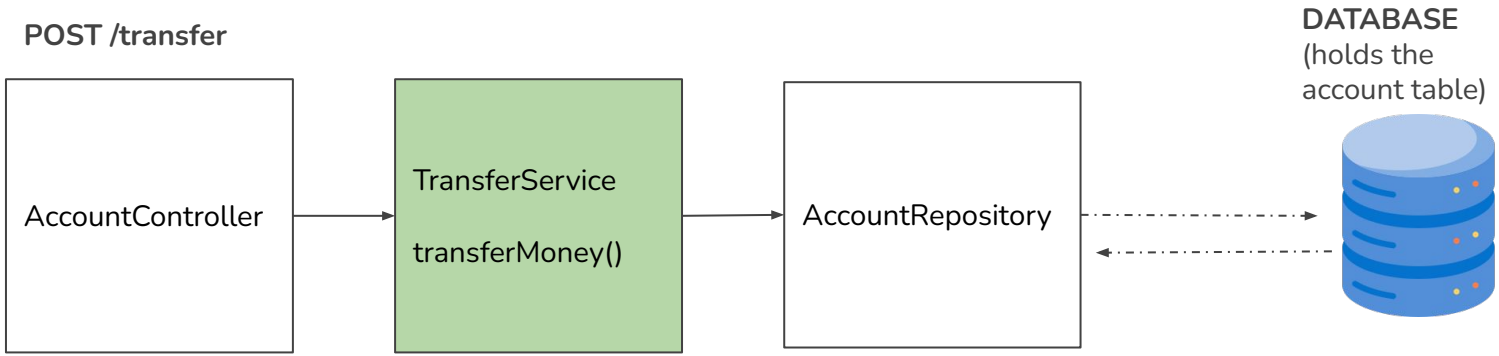
4. Using transactions in Spring apps(4/21)



The transferMoney() method logic implements the steps “withdraw the money from the source account” and “deposit the money in the destination account.” These are mutable operations, so we wrap them in a transaction to make sure that, if either fail, the data is rolled back to the way it was before the use case started.

The repository class implements all the operations with the account table in the database.

4. Using transactions in Spring apps(5/21)



We implement the transfer money use case in a service class and expose this service method through a REST endpoint. The service method uses a repository to access the data in the database and change it. The service method (which implements the business logic) must be wrapped in a transaction to avoid data inconsistencies if problems occur during the method execution.

4. Using transactions in Spring apps(6/21)

We find the example in the project “Exercise-1” We’ll **create a Spring Boot project** and **add the dependencies** to its **pom.xml** file, as presented in the next code snippet. We continue using Spring **JDBC** (as we did in presentation 12) and an **H2 in-memory database**:

4. Using transactions in Spring apps(7/21)

Step 1

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
<dependency>
<groupId>com.h2database</groupId>
<artifactId>h2</artifactId>
<scope>runtime</scope>
</dependency>
```

Step 2

Database table name: “account”
Fields below:
id - The primary key. We define this field as an INT value that self increments.
name - The name of the account’s owner.
amount - The amount of money the owner has in the account.

Step 3

```
create table account (
id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(50) NOT NULL,
amount DOUBLE NOT NULL
);
```

We use a “**schema.sql**” file in the project’s resources folder to create the table. In this file, we write the SQL query to create the table, as presented in the next code snippet:

Step 4

```
INSERT INTO account VALUES (NULL, 'Helen Down', 1000);
INSERT INTO account VALUES (NULL, 'Peter Read', 1000);
```

We also add a “data.sql” file near the “schema.sql” in the resources folder to create two records we’ll use later to test. The “**data.sql**” file contains SQL queries to add two account records to the database.

4. Using transactions in Spring apps(8/21)

The Account class that models the account table

```
public class Account {
private long id;
private String name;
  private BigDecimal amount;
// Omitted getters and setters
}
```

We need a class that models the account table to have a way to refer to the data in our app, so we create a class named Account to model the account records in the database, as shown in the following listing.

4. Using transactions in Spring apps(9/21)

To implement the “transfer money” use case, we need the following capabilities in the repository layer:

- Find the details for an account using the account ID
- Update the amount for a given account

We’ll implement these capabilities as discussed in presentation 10, using **JdbcTemplate**.
For step 1, we implement the method findAccountById(long id), which gets the account ID in a parameter and uses **JdbcTemplate** to get the account details for the account with that ID from the database. For step 2, we implement a method named changeAmount(long id, BigDecimal amount). This method sets the amount it gets as the second parameter to the account with the ID it gets in the first parameter. The next listing shows you the implementation of these two methods.

4. Using transactions in Spring apps(10/21)

Implementing the persistence capabilities in the repository

```java
@Repository
public class AccountRepository {

  private final JdbcTemplate jdbc;

  public AccountRepository(JdbcTemplate jdbc) {
    this.jdbc = jdbc;
  }

  public Account findAccountById(long id) {
    String sql = "SELECT * FROM account WHERE id = ?";
    return jdbc.queryForObject(sql, new AccountRowMapper(), id);
  }
  public void changeAmount(long id, BigDecimal amount) {
    String sql = "UPDATE account SET amount = ? WHERE id = ?";
    jdbc.update(sql, amount, id);
  }
}
```

We add a bean of this class in the **Spring context** using the **@Repository** annotation to later inject this bean where we use it in the service class.

We use constructor dependency injection to get a **JdbcTemplate** object to work with the database.

4. Using transactions in Spring apps(11/21)

Implementing the persistence capabilities in the repository

```
@Repository
public class AccountRepository {

  private final JdbcTemplate jdbc;

  public AccountRepository(JdbcTemplate jdbc) {
    this.jdbc = jdbc;
  }

  public Account findAccountById(long id) {
    String sql = "SELECT * FROM account WHERE id = ?";
    return jdbc.queryForObject(sql, new AccountRowMapper(), id);
  }
  public void changeAmount(long id, BigDecimal amount) {
    String sql = "UPDATE account SET amount = ? WHERE id = ?";
    jdbc.update(sql, amount, id);
  }
}
```

We get the details of an account by sending the **SELECT** query to the **DBMS** using the **JdbcTemplate queryForObject()** method. We also need to provide a **RowMapper** to tell **JdbcTemplate** how to map a row in the result to our model object.

We change the amount of an account by sending an **UPDATE** query to the **DBMS** using the **JdbcTemplate update()** method.

4. Using transactions in Spring apps(12/21)

Mapping the row to a model object instance with a RowMapper

```
public class AccountRowMapper
  implements RowMapper<Account> {
  @Override
  public Account mapRow(ResultSet resultSet, int i)
    throws SQLException {
    Account a = new Account();
    a.setId(resultSet.getInt("id"));
    a.setName(resultSet.getString("name"));
    a.setAmount(resultSet.getBigDecimal("amount"));
    return a;
  }
}
```

We implement the **RowMapper** contract and provide the model class we map the result row into as a generic type.

We implement the **mapRow()** method, which gets the query result as a parameter (shaped as a **ResultSet** object) and returns the **Account** instance we map the current row to.

4. Using transactions in Spring apps(13/13)

Mapping the row to a model object instance with a RowMapper

```
public class AccountRowMapper
  implements RowMapper<Account> {
  @Override
  public Account mapRow(ResultSet resultSet, int i)
    throws SQLException {
    Account a = new Account();
    a.setId(resultSet.getInt("id"));
    a.setName(resultSet.getString("name"));
    a.setAmount(resultSet.getBigDecimal("amount"));
    return a;
  }
}
```

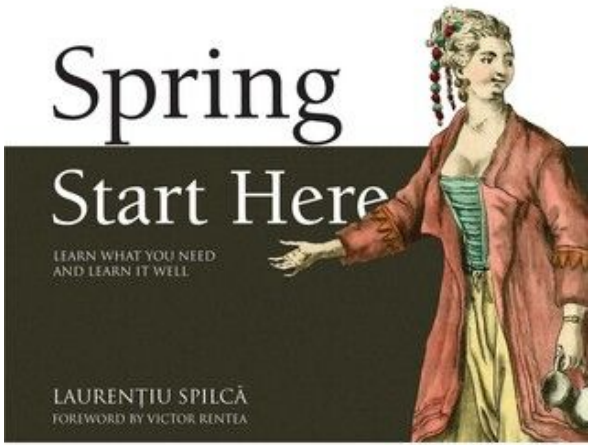
We map the values on the current result row to the **Account's** attributes.

We return the account instance after mapping the result values.

5. Conclusion

- A transaction is a set of operations that change data, which either execute together or not at all. In a real-world scenario, almost any use case should be the subject of a transaction to avoid data inconsistencies.
- If any of the operations fail, the app restores the data to how it was at the beginning of the transaction. When that happens, we say that the transaction rolls back.
- If all the operations succeed, we say the transaction commits, which means the app persists all the changes the use case execution did.
- To implement transactional code in Spring, you use the **@Transactional** annotation. You use the **@Transactional** annotation to mark a method you expect
- Spring to wrap in a transaction. You can also annotate a class with **@Transactional** to tell Spring that any class methods need to be transactional.
- At execution, a Spring aspect intercepts the methods annotated with **@Transactional**. The aspect starts the transaction, and if an exception occurs the aspect rolls back the transaction. If the method doesn't throw an exception, the transaction commits, and the app persists the method's changes.

Resources



MANNING

Reference

1. Spring Start Here
2. Baeldung.com

Thank you!

Presented by

Hamdamboy Urunov & Nizomiddinov Shakhobiddin

(hamdamboy.urunov@gmail.com)

(shohobiddindeveloper@gmail.com)