13 Using transactions in Spring apps

This chapter covers

- What a transaction is
- How Spring manages transactions
- Using transactions in a Spring app

One of the most important things we take into consideration when managing data is to keep accurate data. We don't want specific execution scenarios to end up with wrong or inconsistent data. Let me give you an example. Suppose you implement an application used to share money—an electronic wallet. In this application, a user has accounts where they store their money. You 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 13.1):

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

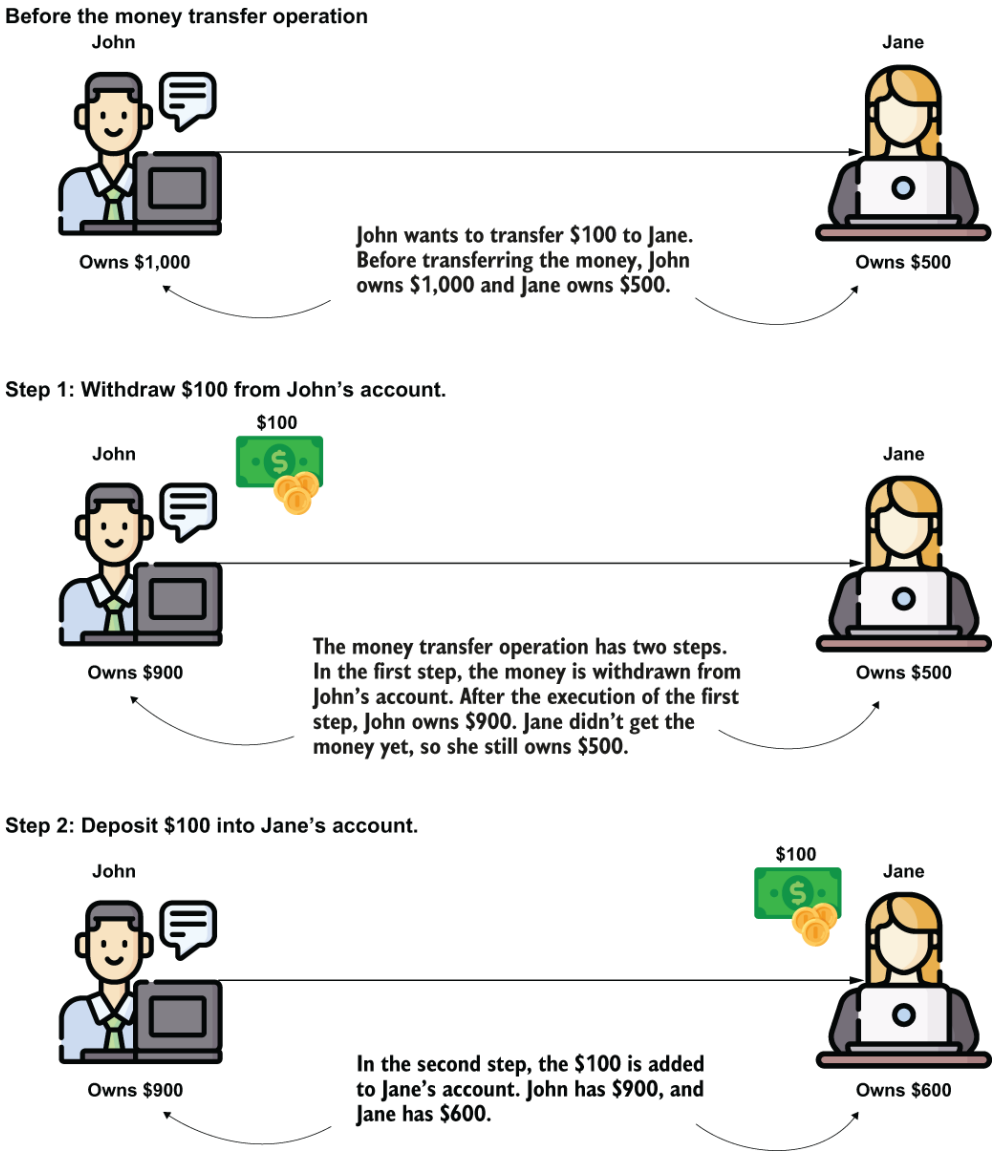

Figure 13.1 An example of a use case. When transferring money from one account to another account, the app executes two operations: it subtracts the transferred money from the first account and adds it to the second account. We'll implement this use case, and we need to make sure its execution won't generate inconsistencies in data.

Both these steps are operations that change data (mutable data operations), and both operations need to be successful to execute the money transfer correctly. But what if the second step encounters a problem and can't complete? If the first finished, but step 2 couldn't complete, the data becomes inconsistent.

Say John sends $100 to Jane. John had $1,000 in his accounts prior to making the transfer, while Jane had $500. After the transfer completes, we expect that John's account will hold $100 less (that is $1,000 - $100 = $900), while Jane will get the $100. Jane should have $500 + $100 = $600.

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.

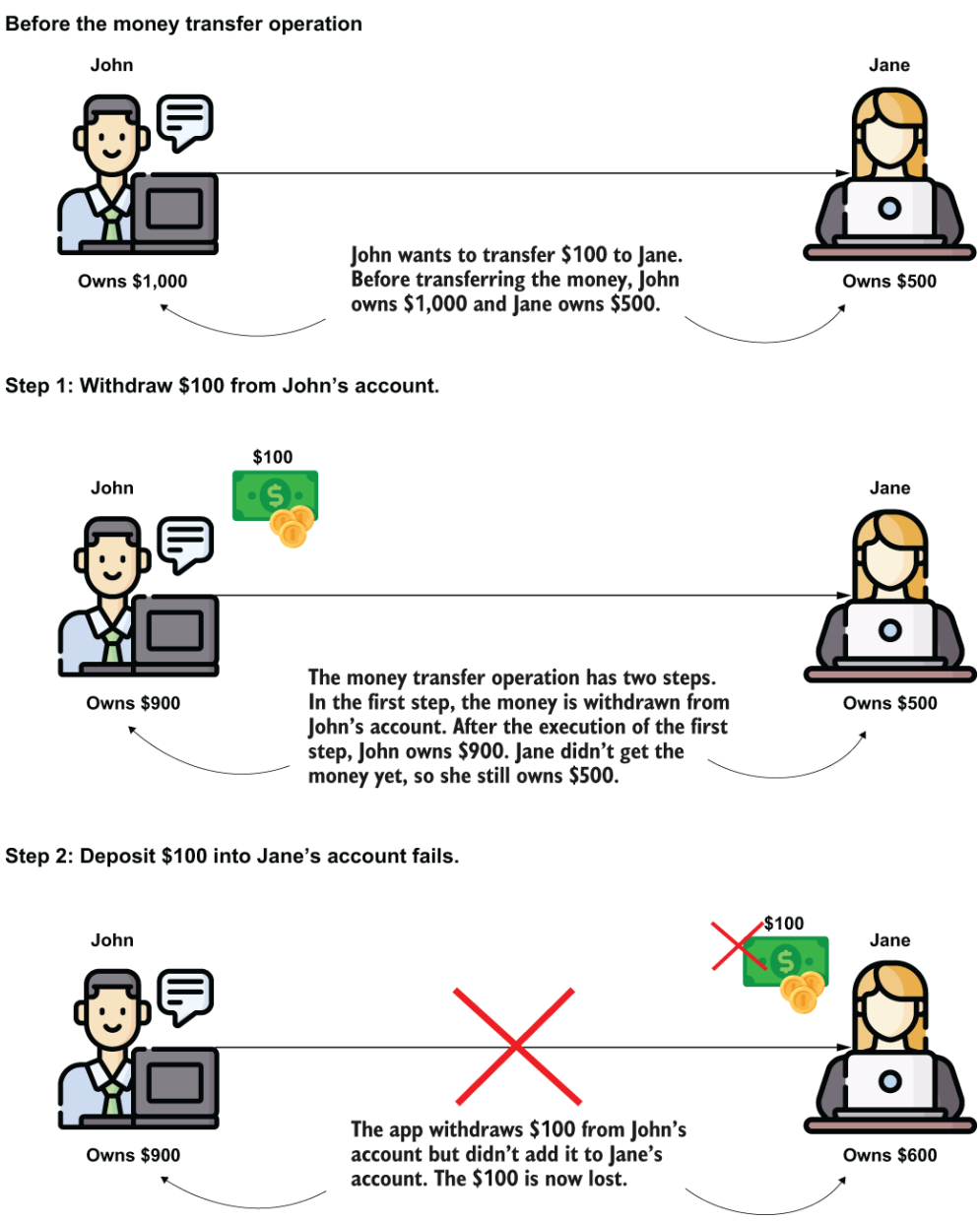

Figure 13.2 If one of the steps of a use case fails, data becomes inconsistent. For the money transfer example, if the operation that subtracts the money from the first accounts succeeds, but the operation that adds it to the destination account fails, money is lost.

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.

13.1 Transactions

In this section, we discuss *transactions*. 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*. 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:

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

We can start a transaction before step 1 and close the transaction after step 2 (figure 13.3). In such a case, if both steps successfully execute, when the transaction ends (after step 2), the app persists the changes made by both steps. We also say, in this case, that the transaction "commits." The "commit" operation happens when the transaction ends and all the steps are successfully executed, so the app persists the data changes.

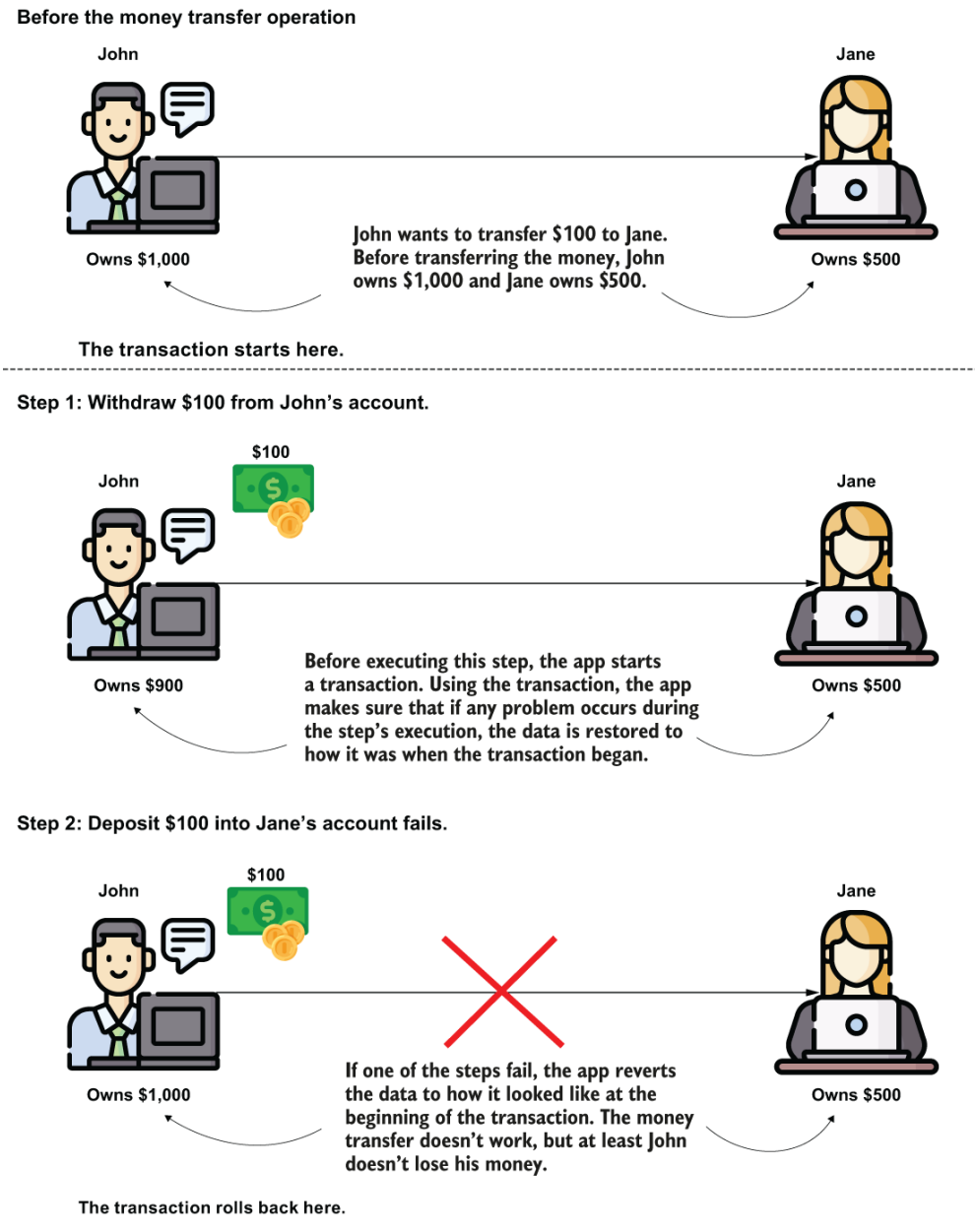

Figure 13.3 A transaction solves possible inconsistencies that could appear if any of the steps of a use case fail. With a transaction, if any of the steps fail, the data is reverted to how it was at the transaction start.

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

If step 1 executes without a problem, but step 2 fails for any reason, the app reverts the changes step 1 made. This operation is named *rollback*.

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.

13.2 How transactions work in Spring

Before showing you how to use transactions in your Spring app, let's discuss how transactions work in Spring and the capabilities the framework offers you for implementing transactional code. In fact, a Spring AOP aspect lies behind the scenes of a transaction. (We discussed how aspects work in chapter 6.)

An aspect is a piece of code that intercepts specific methods' execution in a way that you define. In most cases today, we use annotations to mark the methods whose execution an aspect should intercept and alter. For Spring transactions, things aren't different. To mark a method we want Spring to wrap in a transaction, we use an annotation named `@Transactional`. Behind the scenes, Spring configures an aspect (you don't implement this aspect yourself; Spring provides it) and applies the transaction logic for the operations executed by that method (figure 13.4).

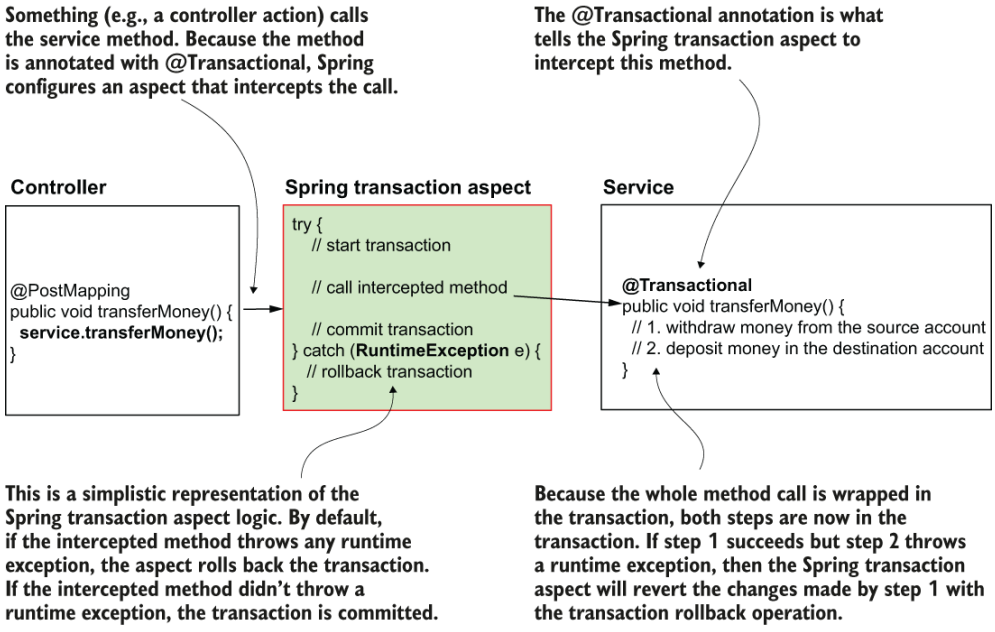

Figure 13.4 When you use the `@Transactional` annotation with a method, an aspect configured by Spring intercepts the method call and applies the transaction logic for that call. The app doesn't persist the changes the method makes if the method throws a runtime exception.

Spring knows to rollback a transaction if the method throws a runtime exception. But I'd like to emphasize the word "throws." When I teach Spring in class, students often understand that it's enough that some oper-

ation inside the `transferMoney()` method throws a runtime exception. But this is not enough! The transactional method should throw the exception further so that the aspect knows it should rollback the changes. If the method treats the exception in its logic and doesn't throw the exception further, the aspect can't know the exception occurred (figure 13.5).

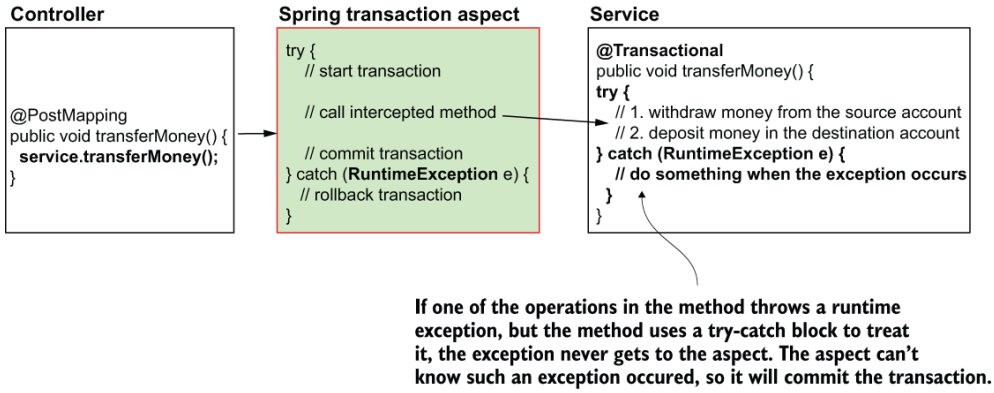

Figure 13.5 If a runtime exception is thrown inside the method, but the method treats the exception and doesn't throw it back to the caller, the aspect won't get this exception and will commit the transaction. When you treat an exception in a transactional method, such as in this case, you need to be aware the transaction won't be rolled back, as the aspect managing the transaction cannot see the exception.

What about checked exceptions in transactions?

Thus far, I've only discussed runtime exceptions. But what about the checked exceptions? Checked exceptions in Java are those exceptions you have to treat or throw; otherwise, your app won't compile. Do they also cause a transaction rollback if a method throws them? By default, no! Spring's default behavior is only to roll back a transaction when it encounters a runtime exception. This is how you'll find transactions used in almost all real-world scenarios.

When you work with a checked exception, you have to add the "throws" clause in the method signature; otherwise, your code won't compile, so you always know when your logic could throw such an exception. For this reason, a situation represented with a checked exception is not an issue that could cause data inconsistency, but is instead a controlled scenario that should be managed by the logic the developer implements.

If, however, you'd like Spring to also roll back transactions for checked exceptions, you can alter Spring's default behavior. The `@Transactional` annotation, which you'll learn to use in section 13.3, has attributes for

defining which exceptions you want Spring to roll back the transactions for.

However, I recommend you always keep your application simple and, unless needed, rely on the framework's default behavior.

13.3 Using transactions in Spring apps

Let's start with an example that teaches you how to use transactions in a Spring app. Declaring a transaction in a Spring app is as easy as using an annotation: `@Transactional`. You use `@Transactional` to mark a method you want Spring to wrap in a transaction. You don't need to do anything else. Spring configures an aspect that intercepts the methods you annotate with `@Transactional`. This aspect starts a transaction and either commits the method's changes if everything went fine or rolls back the changes if any runtime exception occurred.

We'll write an app that stores account details in a database table. Imagine this is the backend of an electronic wallet app you implement. We'll create the capability to transfer money from one account to another. For this use case, we'll need to use a transaction to ensure the data stays consistent if an exception occurs.

The class design of the app we implement is straightforward. We use a table in a database to store the account details (including the money amount). We implement a repository to work with the data in this table, and we implement the business logic (the money transfer use case) in a service class. The service method that implements the business logic is where we'll need to use a transaction. We expose this use case by implementing an endpoint in the controller class. To transfer money from one account to another, someone needs to call this endpoint. Figure 13.6 illustrates the app's class design.

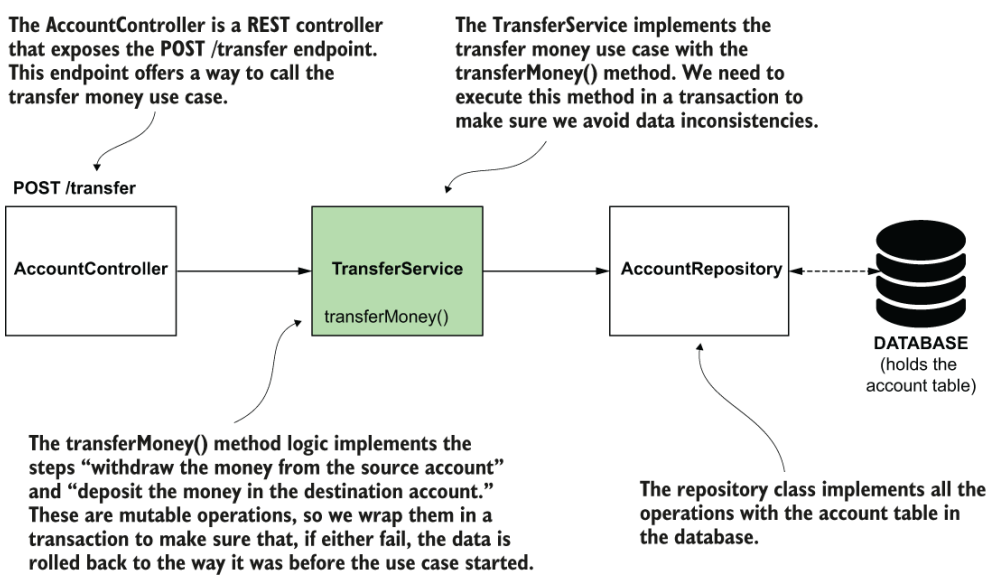

Figure 13.6 We implement the transfer money use case in a service class and expose this service method through a REST endpoint. The service method uses a repository to access the data in the database and change it. The service method (which implements the business logic) must be wrapped in a transaction to avoid data inconsistencies if problems occur during the method execution.

You find the example in the project "sq-ch13-ex1." 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 chapter 12) and an H2 in-memory database:

```
<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>
```

The app works with only one table in a database. We name this table "account," and it has the following fields:

- *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.

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:

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

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. You find these queries in the following code snippet:

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

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.

Listing 13.1 The `Account` class that models the account table

```
public class Account {

  private long id;
  private String name;
  private BigDecimal amount;

  // Omitted getters and setters
}
```

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

1. Find the details for an account using the account ID.
2. Update the amount for a given account.

We’ll implement these capabilities as discussed in chapter 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.

Listing 13.2 Implementing the persistence capabilities in the repository

```
@Repository                                                          ❶
public class AccountRepository {

  private final JdbcTemplate jdbc;

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

  public Account findAccountById(long id) {
    String sql = "SELECT * FROM account WHERE id = ?";               ❸
    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);                                    ❹
  }
}
```

❶ 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.

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

❸ 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.

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

As you learned in chapter 12, when you use `JdbcTemplate` to retrieve data from the database using a SELECT query, you need to provide a `RowMapper` object, which tells `JdbcTemplate` how to map each row of the result from the database to your specific model object. In our case, we need to tell `JdbcTemplate` how to map a row in the result to the `Account` object. The next listing shows you how to implement the `RowMapper` object.

Listing 13.3 Mapping the row to a model object instance with a RowMapper

```
public class AccountRowMapper
  implements RowMapper<Account> {                          ❶

  @Override
  public Account mapRow(ResultSet resultSet, int i)        ❷
    throws SQLException {
    Account a = new Account();                             ❸
    a.setId(resultSet.getInt("id"));                       ❸
    a.setName(resultSet.getString("name"));                ❸
    a.setAmount(resultSet.getBigDecimal("amount"));        ❸
    return a;                                              ❹
  }
}
```

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

❷ 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.

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

❹ We return the account instance after mapping the result values.

To test the app more easily, let's also add the capability to get all the account details from the database, as shown in the following listing. We'll use this capability when verifying that the app works as we expect.

Listing 13.4 Getting all the account records from the database

```
@Repository
public class AccountRepository {

  // Omitted code

  public List<Account> findAllAccounts() {
    String sql = "SELECT * FROM account";
    return jdbc.query(sql, new AccountRowMapper());
  }

}
```

In the service class, we implement the logic for the “transfer money” use case. The `TransferService` class uses the `AccountRepository` class to manage the data in the account table. The logic the method implements is as follows:

1. Get the source and destination account details to find out the amount in both accounts.
2. Withdraw the transferred amount from the first account by setting a new value, which is the account minus the amount to be withdrawn.
3. Deposit the transferred amount into the destination account by setting a new value, the current amount of the account plus the transferred amount.

Listing 13.5 shows you how the `transferMoney()` method of the service class implements this logic. Observe that points 2 and 3 define mutable operations. Both these operations change the persisted data (i.e., they update some account’s amounts). If we don’t wrap them in a transaction, we can get in those cases where the data becomes inconsistent because one of the steps fails.

Fortunately, we only need to use the `@Transactional` annotation to mark the method as transactional and tell Spring it needs to intercept this method’s executions and wrap them in transactions. The following listing shows you the implementation of the money transfer use case logic in the service class.

Listing 13.5 Implementing the money transfer use case in the service class

```
@Service
public class TransferService {

  private final AccountRepository accountRepository;

  public TransferService(AccountRepository accountRepository) {
    this.accountRepository = accountRepository;
  }

  @Transactional                                          ❶
  public void transferMoney(long idSender,
                            long idReceiver,
                            BigDecimal amount) {
    Account sender =                                      ❷
      accountRepository.findAccountById(idSender);        ❷
    Account receiver =                                    ❷
      accountRepository.findAccountById(idReceiver);      ❷

    BigDecimal senderNewAmount =                          ❸
      sender.getAmount().subtract(amount);                ❸
```

```
      BigDecimal receiverNewAmount =                    ❹
        receiver.getAmount().add(amount);               ❹

      accountRepository                                 ❺
       .changeAmount(idSender, senderNewAmount);        ❺

      accountRepository                                 ❻
       .changeAmount(idReceiver, receiverNewAmount);    ❻
    }
  }
```

❶ We use the @Transactional annotation to instruct Spring to wrap the method's calls in transactions.

❷ We get the accounts' details to find the current amount in each account.

❸ We calculate the new amount for the sender account.

❹ We calculate the new amount for the destination account.

❺ We set the new amount value for the sender account.

❻ We set the new amount value for the destination account.

Figure 13.7 visually presents the transaction scope and the steps the `transferMoney()` method executes.

Figure 13.7 The transaction starts just before the service method execution and ends just after the method successfully ended. If the method doesn't throw any runtime exception, the app commits the transaction. If any step causes a runtime exception, the app restores the data to how it was before the transaction started.

Let's also implement a method that retrieves all the accounts. We'll expose this method with an endpoint in the controller class we'll define later. We will use it to check the data was correctly changed when testing the transfer money use case.

Using `@Transactional`

The `@Transactional` annotation can also be applied directly to the class. If used on the class (as presented in the next code snippet), the annotation applies to all the class methods. Often in real-world apps you will find the `@Transactional` annotation used on the class, because the methods of a service class define use cases and, in general, all the use cases need to be transactional. To avoid repeating the annotation on each method, it's easi-

er just to mark the class once. When using `@Transactional` on both the class and the method, the method level's configuration overrides the one on the class:

```
@Service
@Transactional                          ❶
public class TransferService {
  // Omitted code

  public void transferMoney(long idSender,
                            long idReceiver,
                            BigDecimal amount) {

    // Omitted code
  }
}
```

❶ We often use the @Transactional annotation directly with the class. If the class has multiple methods, @Transactional applies to all of them.
The next listing shows you the implementation of the `getAllAccounts()` method, which returns a list of all the database's account records.

Listing 13.6 Implementing a service method that returns all the existing accounts

```
@Service
public class TransferService {

  // Omitted code

  public List<Account> getAllAccounts() {
    return accountRepository.findAllAccounts();
  }
}
```

In the following listing, you find the `AccountController` class's implementation that defines the endpoints that expose the service methods.

Listing 13.7 Exposing the use cases through REST endpoints in the controller class

```
@RestController
public class AccountController {

  private final TransferService transferService;

  public AccountController(TransferService transferService) {
```

```
    this.transferService = transferService;
  }

  @PostMapping("/transfer")                    ❶
  public void transferMoney(
      @RequestBody TransferRequest request     ❷
      ) {
    transferService.transferMoney(             ❸
        request.getSenderAccountId(),
        request.getReceiverAccountId(),
        request.getAmount());
  }

  @GetMapping("/accounts")
  public List<Account> getAllAccounts() {
    return transferService.getAllAccounts();
  }
}
```

❶ We use the HTTP POST method for the /transfer endpoint because it operates changes in the database's data.

❷ We use a request body to get the needed values (source account ID, destination account ID, and amount to be transferred).

❸ We call the service transferMoney() method, the transactional method that implements the transfer money use case.

We use an object of type `TransferRequest` as the `transferMoney()` controller action parameter. The `TransferRequest` object simply models the HTTP request body. Such objects, whose responsibility is to model the data transferred between two apps, are DTOs. The following listing shows the definition of the `TransferRequest` DTO.

Listing 13.8 The `TransferRequest` data transfer object modeling the HTTP request body

```
public class TransferRequest {

  private long senderAccountId;
  private long receiverAccountId;
  private BigDecimal amount;

  // Omitted code
}
```

Start the application, and let's test how the transaction works. We use cURL or Postman to call the endpoint the app exposes. First, let's call the

/accounts endpoint to check how the data looks before executing any transfer money operation. The next snippet shows you the cURL command to use to call the /accounts endpoint:

```
curl http://localhost:8080/accounts
```

Once you run this command, you should find an output in the console similar to the one presented in the next snippet:

```
[
 {"id":1,"name":"Helen Down","amount":1000.0},
 {"id":2,"name":"Peter Read","amount":1000.0}
]
```

We have two accounts in the database (we inserted them earlier in this section when we defined the “data.sql” file). Both Helen and Peter own $1,000 each. Let’s now execute the transfer money use case to transfer $100 from Helen to Peter. In the next code snippet, you find the cURL command you need to run to call the /transfer endpoint to send $100 from Helen to Peter:

```
curl -XPOST -H "content-type:application/json" -d '{"senderAccountId":1,
➥ "receiverAccountId":2, "amount":100}' http://localhost:8080/transfer
```

If you call the /accounts endpoint again, you should observe the difference. After the money transfer operation, Helen has $900, while Peter now has $1,100:

```
curl http://localhost:8080/accounts
```

The result of calling the /accounts endpoint after the money transfer operation is presented in the next snippet:

```
[
 {"id":1,"name":"Helen Down","amount":900.0},
 {"id":2,"name":"Peter Read","amount":1100.0}
]
```

The app is working, and the use case gives the expected result. But where do we prove the transaction really works? The app correctly persists the data when everything goes well, but how do we know the app indeed restores the data if something in the method throws a runtime exception? Should we just trust it does? Of course not!

NOTE One of the most important things I learned about apps is that you should never trust something works unless you tested it properly!

I like to say that until you test any feature of your app, it is in a Schrödinger state. It both works and doesn't work until you prove its state. Of course, this is just a personal analogy I make with an essential concept from quantum mechanics.

Let's test the transaction rolls back as expected when some runtime exception occurs. I duplicated the project "sq-ch13-ex1" in the project "sq-ch13-ex2." In this copy of the project, I add only one line of code that throws a runtime exception at the end of the `transferMoney()` service method, as presented in the following listing.

Listing 13.9 Simulating a problem occurs during the use case execution

```
@Service
public class TransferService {

  // Omitted code

  @Transactional
  public void transferMoney(
   long idSender,
   long idReceiver,
   BigDecimal amount) {

    Account sender = accountRepository.findAccountById(idSender);
    Account receiver = accountRepository.findAccountById(idReceiver);

    BigDecimal senderNewAmount = sender.getAmount().subtract(amount);
    BigDecimal receiverNewAmount = receiver.getAmount().add(amount);

    accountRepository.changeAmount(idSender, senderNewAmount);
    accountRepository.changeAmount(idReceiver, receiverNewAmount);

    throw new RuntimeException("Oh no! Something went wrong!");     ❶
  }

}
```

❶ We throw a runtime exception at the end of the service method to simulate a problem that occurred in the transaction.

Figure 13.8 illustrates the change we made in the `transferMoney()` service method.

Figure 13.8 When the method throws a runtime exception, Spring rolls back the transaction. All the successful changes made on the data are not persisted. The app restores the data to how it was when the transaction started.

We start the application and check the account records by calling the /accounts endpoint, which returns all the accounts in the database:

```
curl http://localhost:8080/accounts
```

Once you run this command, you should find an output in the console similar to the one presented in the next snippet:

```
[
 {"id":1,"name":"Helen Down","amount":1000.0},
 {"id":2,"name":"Peter Read","amount":1000.0}
]
```

As in the previous test, we call the /transfer endpoint to transfer $100 from Helen to Peter using the cURL command, shown in the next snippet:

```
curl -XPOST -H "content-type:application/json" -d '{"senderAccountId":1,
➥ "receiverAccountId":2, "amount":100}' http://localhost:8080/transfer
```

Now, the `transferMoney()` method of the service class throws an exception, resulting in an error 500 in the response sent to the client. You should find this exception in the app's console. The exception's stack trace is similar to the one presented in the next code snippet:

```
java.lang.RuntimeException: Oh no! Something went wrong!
    at
com.example.services.TransferService.transferMoney(TransferService.java:
➥ ~[classes/:na]
    at
com.example.services.TransferService$$FastClassBySpringCGLIB$$338bad6b.i
➥ (<generated>) ~[classes/:na]
    at
org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:218)
➥ ~[spring-core-5.3.3.jar:5.3.3]
```

Let's call the /accounts endpoint again and see if the app changed the accounts:

```
curl http://localhost:8080/accounts
```

Once you run this command, you should find an output in the console similar to the one presented in the next snippet:

```
[
 {"id":1,"name":"Helen Down","amount":1000.0},
 {"id":2,"name":"Peter Read","amount":1000.0}
]
```

You observe the data didn't change even if the exception happens after the two operations that change the amounts in the accounts. Helen should have had $900 and Peter $1,100, but both of them still have the same amounts in their accounts. This result is the consequence of the transaction being rolled back by the app, which causes the data to be restored to how it was at the beginning of the transaction. Even if both mutable steps were executed, when the Spring transaction aspect got the runtime exception, it rolled back the transaction.

Summary

- 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.