14 Implementing data persistence with Spring Data

This chapter covers

- How Spring Data works
- Defining Spring Data repositories
- Using Spring Data JDBC to implement a Spring app's persistence layer

In this chapter, you'll learn to use Spring Data, a Spring ecosystem project that gives you the possibility of implementing a Spring app's persistence layer with minimum effort. As you already know, an application framework's essential role is providing out-of-the-box capabilities that you can directly plug into apps. Frameworks help us save time and also make apps' design easier to understand.

You'll learn to create the app's repositories by declaring interfaces. You'll let the framework provide implementations for these interfaces. You'll literally enable your app to work with a database without implementing the repository yourself and with minimum effort.

We'll start the chapter by discussing how Spring Data works, and in section 14.2 you'll learn how Spring Data integrates into Spring apps. We'll then continue in section 14.3 with a practical example where you'll learn to use Spring Data JDBC to implement an application's persistence layer.

14.1 What Spring Data is

In this section, we discuss what Spring Data is and why we should use this project to implement a Spring app's persistence capabilities. Spring Data is a Spring ecosystem project that simplifies the persistence layer's development by providing implementations according to the persistence technology we use. This way, we only need to write a few lines of code to define the repositories of our Spring app. Figure 14.1 offers a visual representation of Spring Data's place from an app's perspective.

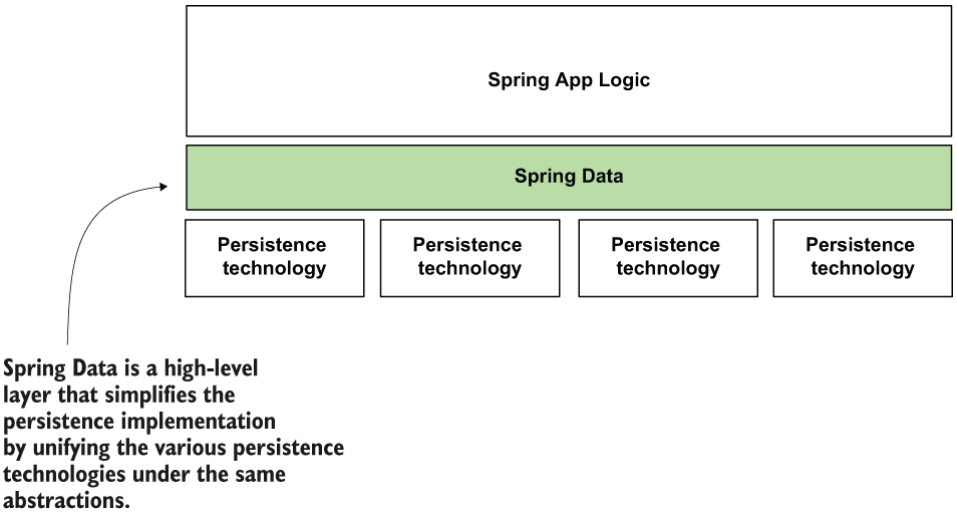

Figure 14.1 The Java ecosystem offers a large number of various persistence technologies. You use each technology in a specific way. Each technology has its own abstractions and class design. Spring Data offers a common abstraction layer over all these persistence technologies to simplify the use of multiple persistence technologies.

Let's see where Spring Data fits in a Spring app. In an app, you have various technologies you can use to work with persisted data. In chapters 12 and 13, we used JDBC, which directly connects to a relational DBMS through a driver manager. But JDBC isn't the only approach you can use to connect to a relational database. Another common way to implement data persistence is using an ORM framework, such as Hibernate. And relational databases aren't the only kind of persisting data technologies. An app might use one of the various NoSQL technologies out there to persist data.

Figure 14.2 shows you some of Spring's alternatives to persist data. Each alternative has its own way of implementing the app's repositories. Sometimes, you even have more options to implement the app's persistence layer for one technology (such as JDBC). For example, with JDBC, you can use `JdbcTemplate`, as you learned in chapter 12, but you could work directly with the JDK interfaces (`Statement`, `PreparedStatement`, `ResultSet`, and so on). Having so many ways to implement the app's persistence capabilities adds complexity.

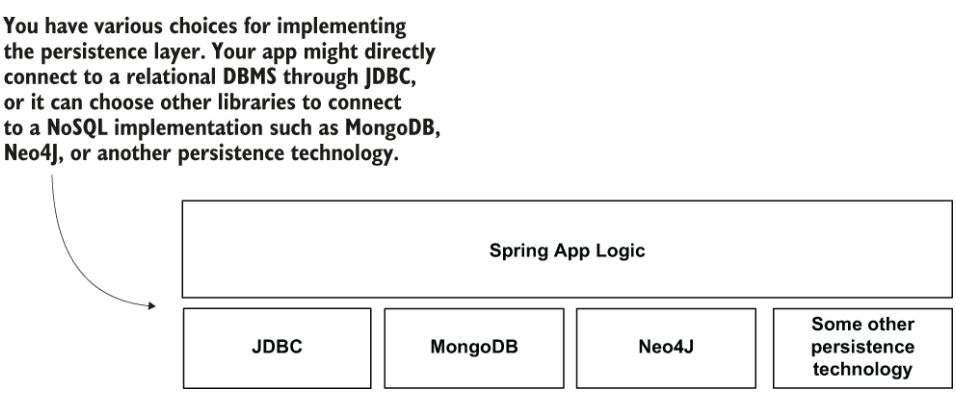

Figure 14.2 Using JDBC to connect to a relational DBMS is not the only choice for implementing an app's persistence layer. In real-world scenarios, you'll use other choices as well, and each way to persist data has its own library and set of APIs you need to learn to use. This variety adds a lot of complexity.

The diagram gets more complicated if we include ORM frameworks such as Hibernate. Figure 14.3 shows Hibernate's place in the scene. Your app could use JDBC directly in various ways, but it could also rely on a framework implemented over JDBC.

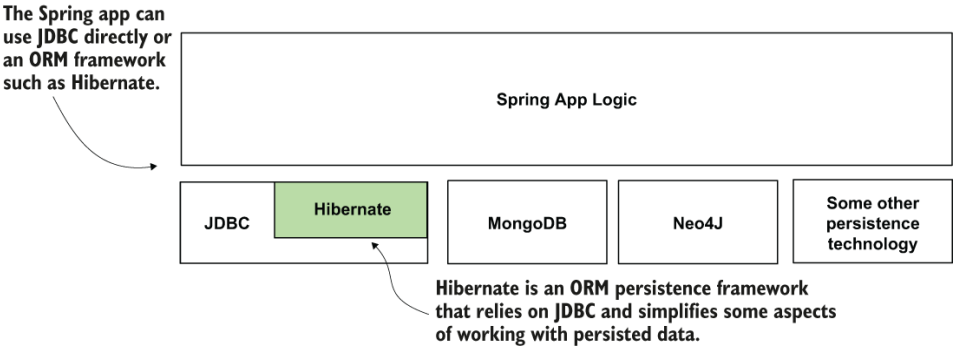

Figure 14.3 Sometimes apps use frameworks built on top of JDBC, such as Hibernate. The variety in choices makes implementing a persistence layer complex. We want to eliminate this complexity from our apps, and, as you'll learn, Spring Data helps us do this.

Don't be worried! You don't need to learn all these at once, and you don't need to know all of them to learn Spring Data. Fortunately, knowing what we already discussed in chapters 12 and 13 on JDBC is enough for a foundation to start learning Spring Data. The reason I made you aware of all these is to demonstrate why Spring Data is so valuable. You might have already asked yourself, "Is there a way we can implement the persistence for all these technologies instead of having to know different approaches for each?" The answer is yes, and Spring Data helps us achieve this goal.

Spring Data simplifies the implementation of the persistence layer by doing the following:

- Providing a common set of abstractions (interfaces) for various persistence technologies. This way, you use a similar approach for implementing the persistence for different technologies.
- Allowing the user to implement the persistence operations using only the abstractions, for which Spring Data provides the implementations. This way, you write less code, so you more quickly implement the app's capabilities. With less written code, the app also becomes easier to understand and maintain.

Figure 14.4 shows Spring Data's position in a Spring app. As you observe, Spring Data is a high-level layer over the various ways to implement the persistence. So, whichever is your choice to implement your app's persistence, if you use Spring Data, you'll write the persistence operations similarly.

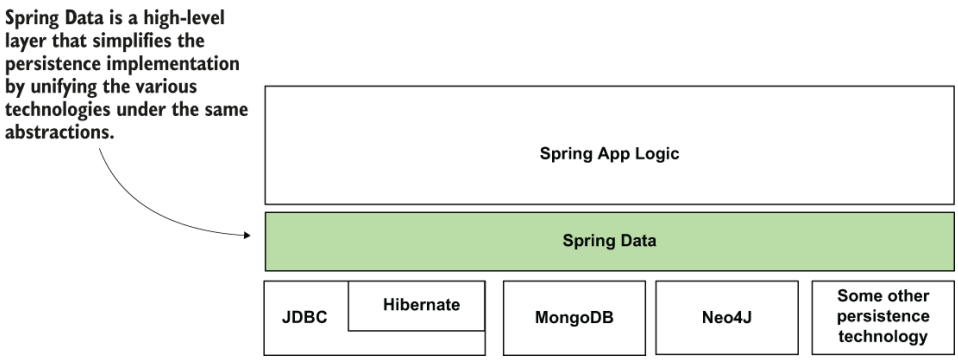

Figure 14.4 Spring Data simplifies the persistence layer implementation by offering a common set of abstractions for various technologies.

14.2 How Spring Data works

In this section, we discuss how Spring Data works and how you'll use it for implementing your Spring app's persistence layer. When developers use the term "Spring Data," they refer in general to all the capabilities this project provides to your Spring app to connect to one persistence technology or another. In an app, generally you use a specific technology: JDBC, Hibernate, MongoDB, or another technology.

The Spring Data project offers different modules for one technology or another. These modules are independent of one another, and you can add them to your project using different Maven dependencies. So, when you implement an app, you don't use *the* Spring Data dependency. There is no such thing as one Spring Data dependency. The Spring Data project provides one Maven dependency for each persistence fashion it supports. For example, you can use the Spring Data JDBC module to connect to the DMBS directly through JDBC, or use the Spring Data Mongo module to connect to a MongoDB database. Figure 14.5 shows what Spring Data looks like using JDBC.

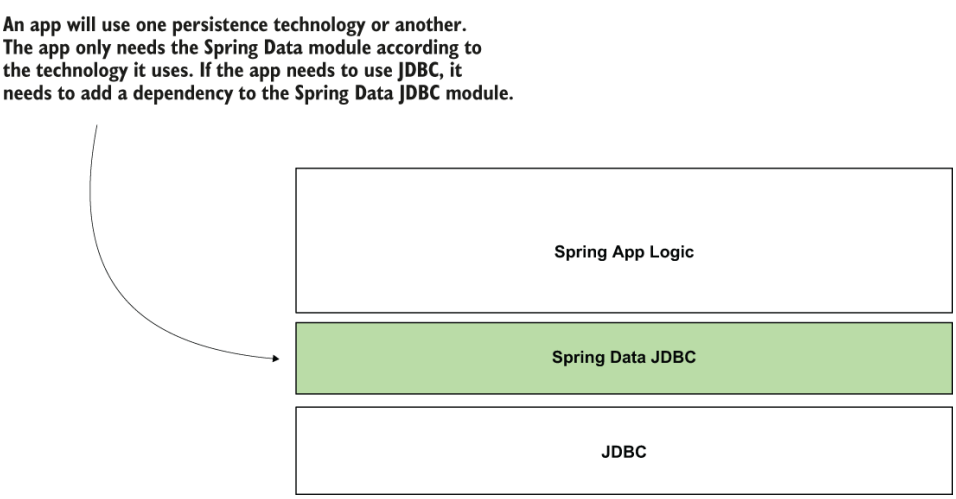

Figure 14.5 If the app uses JDBC, it only needs the part of the Spring Data project that manages persistence through JDBC. The Spring Data module that manages the persistence through JDBC is called Spring Data JDBC. You add this Spring Data module to your app through its own dependency.

You can find the full list of Spring Data modules on Spring Data's official page: **https://spring.io/projects/spring-data.**

Whichever persistence technology your app uses, Spring Data provides a common set of interfaces (contracts) you extend to define the app's persistence capabilities. Figure 14.6 presents the following interfaces:

- `Repository` is the most abstract contract. If you extend this contract, your app recognizes the interface you write as a particular Spring Data repository. Still, you won't inherit any predefined operations (such as adding a new record, retrieving all the records, or getting a record by its primary key). The `Repository` interface doesn't declare any method (it is a marker interface).
- `CrudRepository` is the simplest Spring Data contract that also provides some persistence capabilities. If you extend this contract to define your app's persistence capabilities, you get the simplest operations for creating, retrieving, updating, and deleting records.
- `PagingAndSortingRepository` extends `CrudRepository` and adds operations related to sorting the records or retrieving them in chunks of a specific number (pages).

Figure 14.6 To implement your app's repositories using Spring Data, you extend specific interfaces. The main interfaces that represent Spring Data contracts are `Repository`, `CrudRepository`, and `PagingAndSortingRepository`. You extend one of these contracts to implement your app's persistence capabilities.

NOTE Don't confuse the `@Repository` annotation we discussed in chapter 4 with the Spring Data `Repository` interface. The `@Repository` annotation is the stereotype annotation you use with classes to instruct Spring to add an instance of the annotated class to the application context. This `Repository` interface we discuss in this chapter is specific to Spring Data and, as you'll learn, you extend it or another interface that extends from it to define a Spring Data repository.

Maybe you wonder why Spring Data provides multiple interfaces that extend one another. Why not only one interface with all the operations in it? By implementing multiple contracts that extend each other instead of providing you one "fat" contract with all the operations, Spring Data gives your app the possibility to implement only the operations it needs. This approach is a known principle called *interface segregation*. For example, if your app only needs to use CRUD operations, it extends the `CrudRepository` contract. Your app won't get the operations related to sorting and paging records, making your app simpler (figure 14.7).

Figure 14.7 To create a Spring Data repository, you define an interface that extends one of the Spring Data contracts. For example, if your app only needs CRUD operations, the interface you define as a repository should extend the `CrudRepository` interface. The app adds a bean that implements the contract you define to the Spring context, so any other app components that need to use it can simply inject it from the context.

If your app also needs paging and sorting capabilities over simple CRUD operations, it should extend a more particular contract, the `PagingAndSortingRepository` interface (figure 14.8).

Figure 14.8 If the app needs sorting and paging capabilities, it should extend a more particular contract. The app provides a bean that implements the contract, which can then be injected from any other component that needs to use it.

Some Spring Data modules might provide specific contracts to the technology they represent. For example, using Spring Data JPA, you also can extend the `JpaRepository` interface directly (as presented in figure 14.9). The `JpaRepository` interface is a contract more particular than `PagingAndSortingRepository`. This contract adds operations applicable only when using specific technologies like Hibernate that implement the Jakarta Persistence API (JPA) specification.

Figure 14.9 Spring Data modules that are specific to certain technologies might provide particular contracts that define operations you can apply only with those technologies. When using such technologies, your app most likely will use these specific contracts.

Another example is using a NoSQL technology such as MongoDB. To use Spring Data with MongoDB, you would need to add the Spring Data Mongo module to your app, which also provides a particular contract named `MongoRepository` that adds operations specific to this persistence technology.

When an app uses certain technologies, it extends Spring Data contracts that provide operations particular to that technology. The app could still implement `CrudRepository` if it doesn't need more than the CRUD operations, but these specific contracts usually provide solutions that are more comfortable to use with the specific technology they're made for. In figure 14.10, the `AccountRepository` class (of the app) extends from `JpaRepository` (specific to the Spring Data JPA module).

Figure 14.10 Different Spring Data modules might provide other, more particular contracts. For example, if you use an ORM framework such as Hibernate (which implements the JPA) with Spring Data, you can extend the `JpaRepository` interface, which is a more particular contract that provides operations applicable only when using a JPA implementation, such as Hibernate.

14.3 Using Spring Data JDBC

In this section, we use Spring Data JDBC to implement the persistence layer of a Spring app. We discussed that all you need to do is extend a Spring Data contract, but let's see it in action. In addition to implementing a plain repository, you'll also learn how to create and use custom repository operations.

We'll consider a scenario similar to the one we worked on in chapter 13. The application we build is an electronic wallet managing its users' accounts. A user can transfer money from their account to another account. In this tutorial, we implement the money transfer use case to allow the user to send money from one account to another. The money transfer operation has two steps (figure 14.11):

1. Withdraw a given amount from the sender's account.
2. Deposit the amount in the destination account.

Figure 14.11 The money transfer use case implies two steps. First, the app withdraws the transferred amount from the sender's (John's) account. Second, the app deposits the transferred amount into the receiver's (Jane's) account.

We'll store the account details in a table in the database. To keep the example short and simple and allow you to focus on this section's subject, we'll use an H2 in-memory database (as discussed in chapter 12).

The account table 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

You can find this example in the project "sq-ch14-ex1." The dependencies we need to add to the project (in the pom.xml file) are presented in the next code snippet:

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

❶ We use the Spring Data JDBC module to implement this app's persistence layer.

We add a "schema.sql" file in the Maven project's resources folder to create the account table in the app's H2 in-memory database. This file stores the DDL query needed to create the account 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 need to add a couple of records to the account table. We use these records to test the application later when we finish implementing it. To instruct the app to add a couple of records, we create a "data.sql" file in the Maven project's resource folder. To add two records in the account table, we'll write a couple of `INSERT` statements in the "data.sql" file, as presented in the next code snippet:

```
INSERT INTO account VALUES (NULL, 'Jane Down', 1000);
INSERT INTO account VALUES (NULL, 'John Read', 1000);
```

At the end of the section, we'll demonstrate the app works by transferring $100 from Jane to John. Let's model the account table records with a class named `Account`. We use a field to map each column in the table with the proper type.

Remember that, for decimals, I recommend using `BigDecimal` instead of `double` or `float` to avoid potential issues with the precision in arithmetic operations.

For several operations it provides, such as retrieving data from the database, Spring Data needs to know which field maps the table's primary key. You use the `@Id` annotation, as shown in listing 14.1, to mark the primary key. The following listing shows the `Account` model class.

Listing 14.1 The `Account` class that models the account table records

```
public class Account {

  @Id                  ❶
  private long id;

  private String name;
  private BigDecimal amount;

  // Omitted getters and setters

}
❶ We annotate the attribute that models the primary key with the @Id annot
```

Now that you have a model class, we can implement the Spring Data repository (listing 14.2). We only need CRUD operations for this application, so we'll write an interface that extends the `CrudRepository` interface. All the Spring Data interfaces have two generic types you need to provide:

1. The model class (sometimes named entity) for which you write the repository
2. The primary key field type

Listing 14.2 Defining the Spring Data repository

```
public interface AccountRepository
    extends CrudRepository<Account, Long> {      ❶

}
```

❶ The first generic type value is the type of the model class representing the table. The second is the type of the primary key field.

When you extend the `CrudRepository` interface, Spring Data provides simple operations like getting a value by its primary key, getting all the records from the table, deleting records, and so on. But it can't give you all the possible operations you could implement with SQL queries. In a real-world app, you need custom operations, which need a written SQL query to be implemented. How do you implement a custom operation in a Spring Data repository?

Spring Data makes this aspect so easy that you sometimes don't even need to write a SQL query. Spring Data knows to interpret the method's names based on some naming definition rules and creates the SQL query behind the scenes for you. For example, say you want to write an operation to get all the accounts for a given name. In Spring Data, you can write a method with the following name: `findAccountsByName`.

When the method name starts with "find," Spring Data knows you want to `SELECT` something. Next, the word "Accounts" tells Spring Data what you want to `SELECT`. Spring Data is so smart that I could have even named the method `findByName`. It would still know what to select just because the method is in the `AccountRepository` interface. In this example, I wanted to be more specific and make the operation name clear. After the "By" in the method's name, Spring Data expects to get the query's condition (the `WHERE` clause). In our case, we want to select "ByName," so Spring Data translates this to `WHERE name = ?`.

Figure 14.12 visually represents the relationship between the method's name and the query Spring Data creates behind the scenes.

Figure 14.12 The relationship between the repository's method name and the query Spring Data creates behind the scenes

The following listing shows the definition of the method in the `AccountRepository` interface.

Listing 14.3 Adding a repository operation to get all the accounts with a specified name

```
public interface AccountRepository
  extends CrudRepository<Account, Long> {

  List<Account> findAccountsByName(String name);

}
```

This magic of translating a method's name into a query looks incredible at first sight. However, with experience you realize it's not a silver bullet. It has a few disadvantages, so I always recommend developers explicitly specify the query instead of relying on Spring Data to translate the method's name. The main disadvantages of relying on the method's name are as follows:

- If the operation requires a more complex query, the method's name would be too large and difficult to read.
- If a developer refactors the method's name by mistake, they might affect the app's behavior without realizing it (unfortunately, not all apps are roughly tested, and we need to consider this).
- Unless you have an IDE that offers you hints while writing the method's name, you need to learn the Spring Data's naming rules. Since you already know SQL, learning a set of rules applicable only for Spring Data is not advantageous.
- Performance is affected because Spring Data also has to translate the method name into a query, so the app will initialize slower (the app translates the method names into queries when the app boots).

The simplest way to avoid these problems is using the `@Query` annotation to specify the SQL query that the app will run when you call that method. When you annotate the method `@Query`, it's no longer relevant how you name that method. Spring Data will use the query you provide instead of translating the method's name into a query. The behavior also becomes more performant. The following listing shows you how to use the `@Query` annotation.

Listing 14.4 Using the `@Query` annotation to specify the SQL query for an operation

```
public interface AccountRepository
  extends CrudRepository<Account, Long> {


  @Query("SELECT * FROM account WHERE name = :name")      ❶
  List<Account> findAccountsByName(String name);

}
```

❶ Remember that the parameter's name in the query should be the same as the method parameter's name. There shouldn't be any spaces between the colon (:) and the parameter's name.

You use the `@Query` annotation in the same way to define any query. However, when your query changes data, you also need to annotate the method with the `@Modifying` annotation. If you use `UPDATE`, `INSERT`, or `DELETE`, you also need to annotate the method with `@Modifying`. The following listing shows you how to use `@Query` to define an `UPDATE` query for a repository method.

Listing 14.5 Defining a modifying operation in the repository

```
public interface AccountRepository
  extends CrudRepository<Account, Long> {

  @Query("SELECT * FROM account WHERE name = :name")
  List<Account> findAccountsByName(String name);

  @Modifying                                                    ❶
  @Query("UPDATE account SET amount = :amount WHERE id = :id")
  void changeAmount(long id, BigDecimal amount);

}
```

❶ We annotate the methods that define operations that change data with the @Modifying annotation.

Use DI to get a bean that implements the `AccountRepository` interface wherever you need it in the app. Don't worry that you only wrote the interface. Spring Data creates a dynamic implementation and adds a bean to your app's context. Listing 14.6 shows how the `TransferService` component of the app uses constructor injection to get a bean of type `AccountRepository`. In chapter 5 you learned that Spring is smart and knows that if you requested a DI for a field with an interface type, it needs to find a bean that implements that interface.

Listing 14.6 Injecting the repository in the service class to implement the use case

```
@Service
public class TransferService {

  private final AccountRepository accountRepository;

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

```
}
```

Listing 14.7 shows the implementation of the money transfer use case. We use the `AccountRepository` to get the account details and change the accounts' amounts. We continue to use the `@Transactional` annotation, as you learned in chapter 13, to wrap the logic in a transaction and make sure we don't mess with the data if any of the operations fail.

Listing 14.7 Implementing the transfer money use case

```
@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.findById(idSender)                          ❷
        .orElseThrow(() -> new AccountNotFoundException());         ❷

    Account receiver =                                              ❷
      accountRepository.findById(idReceiver)
        .orElseThrow(() -> new AccountNotFoundException());

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

    BigDecimal receiverNewAmount =                                  ❸
      receiver.getAmount().add(amount);                             ❸

    accountRepository                                               ❹
      .changeAmount(idSender, senderNewAmount);                     ❹

    accountRepository                                               ❹
      .changeAmount(idReceiver, receiverNewAmount);                 ❹
  }

}
```

❶ We wrap the use case logic in a transaction to avoid data inconsistencies if any instruction fails.

❷ We get the sender and receiver's account details.

❸ We calculate the new account amounts by subtracting the transferred value from the sender account and adding it to the destination account.

❹ We change the accounts' amounts in the database.

In the transfer money use case, we used a simple runtime exception class named `AccountNotFoundException`. The next code snippet presents the definition of this class:

```
public class AccountNotFoundException extends RuntimeException {
}
```

Let's add a service method to retrieve all the records from the database and get the account details by the owner's name. We'll use these operations when testing our app. To get all records, we didn't write the method ourselves. Our `AccountRepository` inherits the `findAll()` method from the `CrudRepository` contract, as shown in the following listing.

Listing 14.8 Adding service methods to retrieve account details

```
@Service
public class TransferService {

  // Omitted code

  public Iterable<Account> getAllAccounts() {
    return accountRepository.findAll();            ❶
  }

  public List<Account> findAccountsByName(String name) {
    return accountRepository.findAccountsByName(name);
  }
}
```

❶ AccountRepository inherits this method from the Spring Data CrudRepository interface.

The following listing shows you how the `AccountController` class exposes the money transfer use case through a REST endpoint.

Listing 14.9 Exposing the transfer money use case with a REST endpoint

```
@RestController
public class AccountController {

  private final TransferService transferService;
```

```
  public AccountController(TransferService transferService) {
    this.transferService = transferService;
  }

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

}
```

❶ We get the sender and destination account IDs and the transferred amount in the HTTP request body.

❷ We call the service to execute the money transfer use case.

The next code snippet presents the `TransferRequest` DTO implementation the /transfer endpoint uses to map the HTTP request body:

```
public class TransferRequest {

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

  // Omitted getters and setters
}
```

In the next listing, we implement an endpoint to fetch the records from the database.

Listing 14.10 Implementing an endpoint to retrieve account details

```
@RestController
public class AccountController {

  // Omitted code

  @GetMapping("/accounts")
  public Iterable<Account> getAllAccounts(                  ❶
      @RequestParam(required = false) String name
  ) {
    if (name == null) {                                     ❷
      return transferService.getAllAccounts();
    } else {                                                ❸
      return transferService.findAccountsByName(name);
```

```
    }
  }

}
```

❶ We use an optional request parameter to get the name for which we want to return the account details.

❷ If no name is provided in the optional request parameter, we return all the account details.

❸ If a name is provided in the request parameter, we only return the account details for the given name.

We start the application and check the account records by calling the /accounts endpoint, which returns all 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":"Jane Down","amount":1000.0},
 {"id":2,"name":"John Read","amount":1000.0}
]
```

We call the /transfer endpoint to transfer $100 from Jane to John 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
```

If you call the /accounts endpoint again, you should observe the difference. After the money transfer operation, Jane has only $900, while John 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":"Jane Down","amount":900.0},
 {"id":2,"name":"John Read","amount":1100.0}
]
```

You can also request to see only Jane's accounts if you use the name query parameter with the /accounts endpoint, as presented in the next snippet:

```
curl http://localhost:8080/accounts?name=Jane+Down
```

As presented in the next snippet, in the response body for this cURL command, you'll only get Jane's accounts:

```
[
    {
        "id": 1,
        "name": "Jane Down",
        "amount": 900.0
    }
]
```

Summary

- Spring Data is a Spring ecosystem project that helps us more easily implement a Spring app's persistence layer. Spring Data provides an abstraction layer over multiple persistence technologies and facilitates the implementation by providing a common set of contracts.
- With Spring Data, we implement repositories through interfaces that extend standard Spring Data contracts:
  - `Repository`, which doesn't provide any persistence operation
  - `CrudRepository`, which provides simple `CREATE`, `READ`, `UPDATE`, `DELETE` (CRUD) operations
  - `PagingAndSortingRepository`, which extends `CrudRepository` and adds operations for the pagination and sorting of the fetched records
- When using Spring Data, you choose a certain module according to the persistence technology your app uses. For example, if your app connects to the DBMS through JDBC, your app needs the Spring Data JDBC module, while if your app uses a NoSQL implementation such as MongoDB, it needs the Spring Data Mongo module.
- When extending a Spring Data contract, your app inherits and can use the operations defined by that contract. However, your app can define custom operations with methods in the repository interfaces.
- You use the `@Query` annotation with the Spring Data repository method to define the SQL query your app executes for that specific operation.
- If you declare a method and don't explicitly specify a query with the `@Query` annotation, Spring Data will translate the method's name into a SQL query. The method name needs to be defined based on Spring Data rules to understand and translate it into the correct query. If Spring Data cannot solve the method name, the application fails to start and throws an exception.

- It is preferable to use the `@Query` annotation and avoid relying on Spring Data to translate the method name into the query. Using the name translation approach could come with difficulties:
  - It creates long and difficult-to-read method names for more complex operations, which affect the app's maintainability.
  - It slows down the app's initialization because the app needs now to also translate the method names.
  - You need to learn the Spring Data method name convention.
  - It runs the risk of affecting the app's behavior by an incorrect refactor of the method name.
- Any operation that changes data (e.g., executes `INSERT`, `UPDATE`, or `DELETE` queries) must be annotated with the `@Modifying` annotation to instruct Spring Data that the operation changes data records.