Spring Transaction Management: @Transactional In-Depth

Last updated on December 17, 2019 - 19 comments

You can use this guide to get a simple and practical understanding of how Spring's transaction management with the @Transactional annotation works.

The only prerequisite? You need to have a rough idea about ACID, i.e. what database transactions are and why to use them. Also, distributed transactions or reactive transactions are not covered here, though the general principles, in terms of Spring, still apply.

Introduction

In this guide you are going to learn about the main pillars of Spring core's *transaction abstraction framework* (a confusing term, isn't it?) - described with a lot of code examples:

- @Transactional (Declarative Transaction Management) vs Programmatic Transaction Management.
- Physical vs Logical transactions.
- Spring @Transactional and JPA / Hibernate integration.
- Spring @Transactional and Spring Boot or Spring MVC integration.
- Rollbacks, Proxies, Common Pitfalls and much more.

As opposed to, say, the official Spring documentation, this guide won't confuse you by diving right into the topic *Spring-first*.

Instead you are going to learn Spring transaction management the *unconventional way*: From the ground up, step by step. This means, starting with plain old JDBC transaction management.

Why?

Because everything that Spring does is *based on* these very JDBC basics. And you'll save a ton of time with Spring's @Transactional annotation later, if you grasp these basics.

How plain JDBC Transaction Management works

If you are thinking of skipping this section, without knowing JDBC transactions inside-out: **don't**.

How to start, commit or rollback JDBC transactions

The first important take-away is this: It does not matter if you are using Spring's @Transactional annotation, plain Hibernate, jOOQ or any other database library.

In the end, they *all do the very same thing* to open and close (let's call that 'manage') database transactions. Plain JDBC transaction management code looks like this:

```
import java.sql.Connection;

Connection connection = dataSource.getConnection(); // (1)

try (connection) {
    connection.setAutoCommit(false); // (2)
    // execute some SQL statements...
    connection.commit(); // (3)

} catch (SQLException e) {
    connection.rollback(); // (4)
}
```

1. You need a connection to the database to start transactions. DriverManager.getConnection(url, user, password) would work as well, though in most enterprise-y applications you will have a data source configured and get connections from that.
2. This is the **only** way to start a database transaction in Java, even though the name might sound a bit off. *setAutoCommit(true)* wraps every single SQL statement in its own transaction and *setAutoCommit(false)* is the opposite: You are the master of the transaction.
3. Let's commit our transaction…
4. Or, rollback our changes, if there was an exception.

Yes, these 4 lines are (oversimplified) everything that Spring does whenever you are using the @Transactional annotation. In the next chapter you'll find out how that works. But before we go there, there's a tiny bit more you need to learn.

(A quick note for smarty-pants: Connection pool libraries like HikariCP might toggle the autocommit mode automatically for you, depending on the configuration. But that is an advanced topic.)

How to use JDBC isolation levels and savepoints

If you already played with Spring's @Transactional annotation you might have encountered something like this:

```
@Transactional(propagation=TransactionDefinition.NESTED,
               isolation=TransactionDefinition.ISOLATION_READ_UNCOMMITTED)
```

We will cover nested Spring transactions and isolation levels later in more detail, but again it helps to know that these parameters all boil down to the following, basic JDBC code:

```
import java.sql.Connection;
```

```
// isolation=TransactionDefinition.ISOLATION_READ_UNCOMMITTED

connection.setTransactionIsolation(Connection.TRANSACTION_READ_UNCOMMITTED); // (1)

// propagation=TransactionDefinition.NESTED

Savepoint savePoint = connection.setSavepoint(); // (2)
...
connection.rollback(savePoint);
```

1. This is how Spring sets isolation levels on a database connection. Not exactly rocket science, is it?
2. Nested transactions in Spring are just JDBC / database savepoints. If you don't know what a savepoint is, have a look at this tutorial, for example. Note that savepoint support is dependent on your JDBC driver/database.

Recommended: Practice JDBC basics

You can find a ton of code examples and exercises on plain JDBC connections and transactions in the *Plain JDBC* chapter of this Java database e-book.

How Spring's or Spring Boot's Transaction Management works

As you now have a good JDBC transaction understanding, let's have a look at how plain, core Spring manages transactions. Everything here applies 1:1 to Spring Boot and Spring MVC, but more about that a bit later..

What actually *is* Spring's transaction management or its (rather confusingly named) transaction abstraction framework?

Remember, transaction management simply means: How does Spring start, commit or rollback JDBC transactions? Does this sound in any way familiar from above?

Here's the catch: Whereas with plain JDBC you only have one way (setAutocommit(false)) to manage transactions, Spring offers you many different, more convenient ways to achieve the same.

How to use Spring's Programmatic Transaction Management?

The first, but rather sparingly used way to define transactions in Spring is programmatically: Either through a TransactionTemplate or directly through the PlatformTransactionManager. Code-wise, it looks like this:

```
@Service
public class UserService {

    @Autowired
    private TransactionTemplate template;

    public Long registerUser(User user) {
        Long id = template.execute(status ->  {
            // execute some SQL that e.g.
            // inserts the user into the db and returns the autogenerated id
            return id;
        });
    }
}
```

Compared with the plain JDBC example:

- You do not have to mess with opening or closing database connections yourself (try-finally). Instead you use Transaction Callbacks.
- You also do not have to catch SQLExceptions, as Spring converts these exceptions to runtime exceptions for you.
- And you have better integration into the Spring ecosystem. TransactionTemplate will use a TransactionManager internally, which will use a data source. All are beans that you have to specify in your Spring context configuration, but then don't have to worry about anymore later on.

While this counts as a minor improvement, programmatic transaction management is not what Spring's transaction framework mainly is about. Instead, it's all about *declarative transaction management*. Let's find out what that is.

How to use Spring's XML Declarative Transaction Management?

Back in the day, when XML configuration was the norm for Spring projects, you could configure transactions directly in XML. Apart from a couple of legacy, enterprise projects, you won't find this approach anymore in the wild, as it has been superseded with the much simpler @Transactional annotation.

We will not go into detail on XML configuration in this guide, but you can use this example as a starting point to dive deeper into it - if needed (taken straight from the official Spring documentation):

```
<!-- the transactional advice (what 'happens'; see the <aop:advisor/> bean below) -->
    <tx:advice id="txAdvice" transaction-manager="txManager">
        <!-- the transactional semantics... -->
        <tx:attributes>
            <!-- all methods starting with 'get' are read-only -->
            <tx:method name="get*" read-only="true"/>
            <!-- other methods use the default transaction settings (see below) -->
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>
```

You are specifying an AOP advice (Aspect Oriented Programming) with the above XML block, that you can then apply to your UserService bean like so:

```
<aop:config>
    <aop:pointcut id="userServiceOperation" expression="execution(* x.y.service.UserService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="userServiceOperation"/>
</aop:config>

<bean id="userService" class="x.y.service.UserService"/>
```

Your UserService bean would then look like this:

```
public class UserService {

    public Long registerUser(User user) {
        // execute some SQL that e.g.
        // inserts the user into the db and retrieves the autogenerated id
        return id;
    }
}
```

From a Java code perspective, this declarative transaction approach looks a lot simpler than the programmatic approach. But it leads to a lot of complicated, verbose XML, with the pointcut and advisor configurations.

So, this leads to the question: Is there a better way for declarative transaction management instead of XML? Yes, there is: The @Transactional annotation.

How to use Spring's @Transactional annotation (Declarative Transaction Management)

Now let's have a look at what modern Spring transaction management usually looks like:

```
public class UserService {

    @Transactional
    public Long registerUser(User user) {
       // execute some SQL that e.g.
        // inserts the user into the db and retrieves the autogenerated id
        // userDao.save(user);
        return id;
    }
}
```

How is this possible? There is no more XML configuration and there's also no other code needed. Instead, you now need to do two things:

- Make sure that your Spring Configuration is annotated with the @EnableTransactionManagement annotation (In Spring Boot this will be done *automatically for you*).
- Make sure you specify a transaction manager in your Spring Configuration (this you need to do anyway).
- And then Spring is smart enough to transparently handle transactions for you: Any bean's *public* method you annotate with the @Transactional annotation, will execute *inside a database transaction* (note: there are some pitfalls).

So, to get the @Transactional annotation working, all you need to do is this:

```
@Configuration
@EnableTransactionManagement
public class MySpringConfig {

    @Bean
    public PlatformTransactionManager txManager() {
        return yourTxManager; // more on that later
    }

}
```

Now, when I say Spring transparently handles transactions for you. What does that *really mean*?

Armed with the knowledge from the JDBC transaction example, the @Transactional UserService code above translates (simplified) directly to this:

```
public class UserService {

    public Long registerUser(User user) {
        Connection connection = dataSource.getConnection(); // (1)
        try (connection) {
            connection.setAutoCommit(false); // (1)

            // execute some SQL that e.g.
            // inserts the user into the db and retrieves the autogenerated id
            // userDao.save(user); <(2)

            connection.commit(); // (1)
        } catch (SQLException e) {
            connection.rollback(); // (1)
        }
    }
}
```

1. This is all just standard opening and closing of a JDBC connection. That's what Spring's transactional annotation does for you automatically, without you having to write it explicitly.
2. This is your own code, saving the user through a DAO or something similar.

This example might look a bit *magical*, but let's have a look at how Spring inserts this connection code for you.

CGlib & JDK Proxies - @Transactional under the covers

Spring cannot really rewrite your Java class, like I did above, to insert the connection code (unless you are using advanced techniques like bytecode weaving, but we are ignoring that for now).

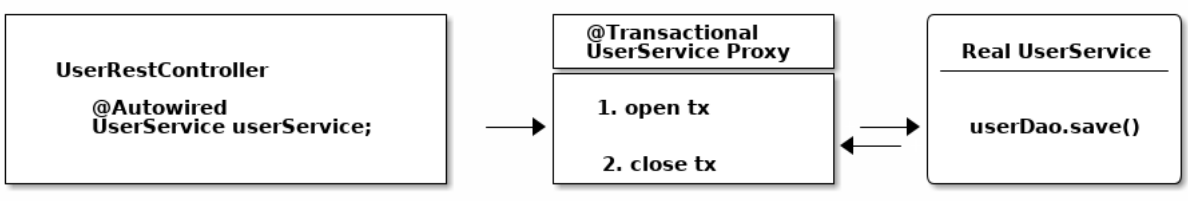
Your registerUser() method really just calls userDao.save(user), there's no way to change that on the fly.

But Spring has an advantage. At its core, it is an IoC container. It instantiates a UserService for you and makes sure to autowire that UserService into any other bean that needs a UserService.

Now whenever you are using @Transactional on a bean, Spring uses a tiny trick. It does not just instantiate a UserService, but also a transactional *proxy* of that UserService.

It does that through a method called *proxy-through-subclassing* with the help of the Cglib library. There are also other ways to construct proxies (like Dynamic JDK proxies), but let's leave it at that for the moment.

Let's see proxies in action in this picture:



As you can see from that diagram, the proxy has one job.

- Opening and closing database connections/transactions.
- And then delegating to the *real UserService*, the one you wrote.
- And other beans, like your UserRestController will never know that they are talking to a proxy, and not the *real* thing.

Quick Exam

Have a look at the following source code and tell me what *type* of UserService Spring automatically constructs, assuming it is marked with @Transactional or has a @Transactional method.

```
@Configuration
@EnableTransactionManagement
public static class MyAppConfig {

    @Bean
    public UserService userService() {  // (1)
        return new UserService();
    }
}
```

1. Correct. Spring constructs a dynamic CGLib proxy of your UserService class here that can open and close database transactions for you. You or any other beans won't even notice that it is not *your* UserService, but a proxy wrapping your UserService.

For what do you need a Transaction Manager (like PlatformTransactionManager)?

Now there's only one crucial piece of information missing, even though we have mentioned it a couple of times already.

Your UserService gets proxied on the fly, and the proxy manages transactions for you. But it is not the proxy itself handling all this transactional state (open, commit, close), the proxy delegates that work to a *transaction manager*.

Spring offers you a PlatformTransactionManager / TransactionManager interface, which, by default, comes with a couple of handy implementations. One of them is the datasource transaction manager.

It does exactly what you did so far to manage transactions, but first, let's look at the needed Spring configuration:

```
@Bean
public DataSource dataSource() {
    return new MysqlDataSource(); // (1)
}

@Bean
public PlatformTransactionManager txManager() {
    return new DataSourceTransactionManager(dataSource()); // (2)
}
```

1. You create a database-specific or connection-pool specific datasource here. MySQL is being used for this example.
2. Here, you create your transaction manager, which needs a data source to be able to manage transactions.

Simple as. All transaction managers then have methods like "doBegin" (for starting a transaction) or "doCommit", which look like this - taken straight from Spring's source code and simplified a bit:

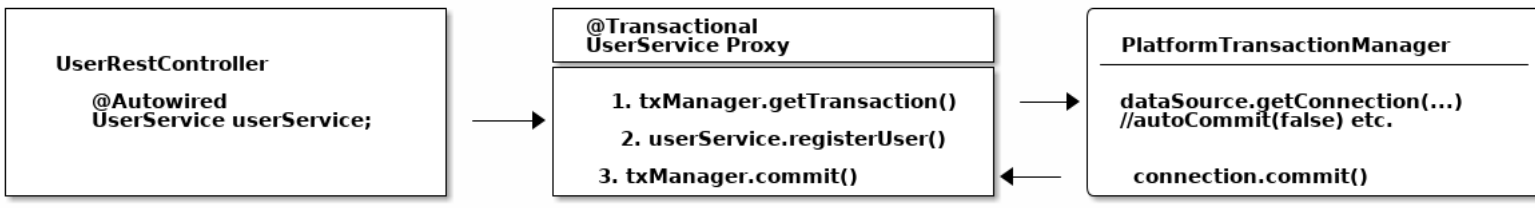
```
public class DataSourceTransactionManager implements PlatformTransactionManager {

    @Override
    protected void doBegin(Object transaction, TransactionDefinition definition) {
        Connection newCon = obtainDataSource().getConnection();
        // ...
        con.setAutoCommit(false);
        // yes, that's it!
    }

    @Override
    protected void doCommit(DefaultTransactionStatus status) {
        // ...
        Connection connection = status.getTransaction().getConnectionHolder().getConnection();
        try {
            con.commit();
        } catch (SQLException ex) {
            throw new TransactionSystemException("Could not commit JDBC transaction", ex);
        }
    }
}
```

So, the datasource transaction manager uses *exactly* the same code that you saw in the JDBC section, when managing transactions.

With this in mind, let's extend our picture from above:



To sum things up:

1. If Spring detects the @Transactional annotation on a bean, it creates a dynamic proxy of that bean.
2. The proxy has access to a transaction manager and will ask it to open and close transactions / connections.
3. The transaction manager itself will simply do what you did in the plain Java section: Manage a good, old JDBC connection.

What is the difference between physical and logical transactions?

Imagine the following two transactional classes.

```
@Service
public class UserService {

    @Autowired
    private InvoiceService invoiceService;

    @Transactional
    public void invoice() {
        invoiceService.createPdf();
        // send invoice as email, etc.
    }
}

@Service
public class InvoiceService {

    @Transactional
    public void createPdf() {
        // ...
    }
}
```

UserService has a transactional invoice() method. Which calls another transactional method, createPdf() on the InvoiceService.

Now in terms of database transactions, this should really just be **one** database transaction. (Remember: *getConnection(). setAutocommit(false). commit().*) Spring calls this *physical transaction*, even though this might sound a bit confusing at first.

From Spring's side however, there's two *logical transactions* happening: First in UserService, the other one in InvoiceService. Spring has to be smart enough to know that both @Transactional methods, should use the same *underlying, physical* database transaction.

How would things be different, with the following change to InvoiceService?

```
@Service
public class InvoiceService {

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void createPdf() {
        // ...
    }
}
```

Changing the propagation mode to requires_new is telling Spring that createPDF() needs to execute in its own transaction, independent of any other, already existing transaction. Thinking back to the plain Java section of this guide, did you see a way to "split" a transaction in half? Neither did I.

Which basically means your code will open **two** (physical) connections/transactions to the database. (Again: *getConnection() x2. setAutocommit(false) x2. commit() x2*) Spring now has to be smart enough that the *two logical transactional* pieces (invoice()/createPdf()) now also map to two *different, physical* database transactions.

So, to sum things up:

- Physical Transactions: Are your actual JDBC transactions.
- Logical Transactions: Are the (potentially nested) @Transactional-annotated (Spring) methods.

This leads us to covering propagation modes in more detail.

What are @Transactional Propagation Levels used for?

When looking at the Spring source code, you'll find a variety of propagation levels or modes that you can plug into the @Transactional method.

```
@Transactional(propagation = Propagation.REQUIRED)

// or

@Transactional(propagation = Propagation.REQUIRES_NEW)
// etc
```

The full list:

- REQUIRED
- SUPPORTS
- MANDATORY
- REQUIRES_NEW
- NOT_SUPPORTED
- NEVER
- NESTED

Exercise:

In the plain Java section, I showed you *everything* that JDBC can do when it comes to transactions. Take a minute to think about what every single Spring propagation mode at the end *REALLY* does to your datasource or rather, your JDBC connection.

Then have a look at the following answers.

Answers:

- **Required (default)**: My method needs a transaction, either open one for me or use an existing one → *getConnection(). setAutocommit(false). commit()*.
- **Supports**: I don't really care if a transaction is open or not, i can work either way → nothing to do with JDBC
- **Mandatory**: I'm not going to open up a transaction myself, but I'm going to cry if no one else opened one up → nothing to do with JDBC
- **Require_new:** I want my completely own transaction → *getConnection(). setAutocommit(false). commit()*.
- **Not_Supported:** I really don't like transactions, I will even try and suspend a current, running transaction → nothing to do with JDBC

- **Never:** I'm going to cry if someone else started up a transaction → nothing to do with JDBC
- **Nested:** It sounds so complicated, but we are just talking savepoints! → *connection.setSavepoint()*

As you can see, most propagation modes really have nothing to do with the database or JDBC, but more with how you structure your program with Spring and how/when/where Spring expects transactions to be there.

Look at this example:

```
public class UserService {

     @Transactional(propagation = Propagation.MANDATORY)
     public void myMethod() {
        // execute some sql
     }

}
```

In this case, Spring will *expect* a transaction to be open, whenever you call myMethod() of the UserService class. It *does not* open one itself, instead, if you call that method without a pre-existing transaction, Spring will throw an exception. Keep this in mind as additional points for "logical transaction handling".

What are @Transactional Isolation Levels used for?

This is almost a trick question at this point, but what happens when you configure the @Transactional annotation like so?

```
@Transactional(isolation = Isolation.REPEATABLE_READ)
```

Yes, it does simply lead to this:

```
connection.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ);
```

Database isolation levels are, however, a complex topic, and you should take some time to fully grasp them. A good start is the official Postgres Documentation and their section on isolation levels.

Also note, that when it comes to switching isolation levels *during* a transaction, you **must** make sure to consult with your JDBC driver/database to understand which scenarios are supported and which not.

The most common @Transactional pitfall

There is one pitfall that Spring beginners usually run into. Have a look at the following code:

```
@Service
public class UserService {

    @Transactional
    public void invoice() {
        createPdf();
        // send invoice as email, etc.
    }

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void createPdf() {
        // ...
    }
}
```

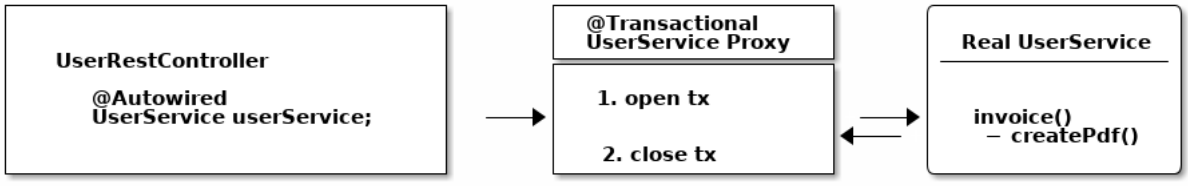
You have a UserService class with a transactional invoice method. Which calls createPDF(), which is also transactional.

How many physical transactions would you expect to be open, once someone calls invoice()?

Nope, the answer is not two, but one. Why?

Let's go back to the proxies' section of this guide. Spring creates that transactional UserService proxy for you, but once you are inside the UserService class and call other inner methods, there is no more proxy involved. This means, no new transaction for you.

Let's have a look at it with a picture:



There's some tricks (like self-injection), which you can use to get around this limitation. But the main takeaway is: always keep the proxy transaction boundaries in mind.

How to use @Transactional with Spring Boot or Spring MVC

So far, we have only talked about plain, core Spring. But what about Spring Boot? Or Spring Web MVC? Do they handle transactions any differently?

The short answer is: No.

With either frameworks (or rather: *all frameworks* in the Spring ecosystem), you will *always* use the @Transactional annotation, combined with a transaction manager and the @EnableTransactionManagement annotation. There is no other way.

The only difference with Spring Boot is, however, that it automatically sets the @EnableTransactionManagement annotation and creates a PlatformTransactionManager for you - with its JDBC auto-configurations. Learn more about auto-configurations here.

Recommended: Practice Spring Transactions

You can find a ton of code examples and exercises on Spring transactions in the *Spring transactions* chapter of this Java database e-book.

How Spring handles rollbacks (and default rollback policies)

The section on Spring rollbacks will be handled in the next revision of this guide.

How Spring and JPA / Hibernate Transaction Management works

The goal: Syncing Spring's @Transactional and Hibernate / JPA

At some point, you will want your Spring application to integrate with another database library, such as Hibernate (a popular JPA-implementation) or Jooq etc.

Let's take plain Hibernate as an example (note: it does not matter if you are using Hibernate directly,or Hibernate via JPA).

Rewriting the UserService from before to Hibernate would look like this:

```
public class UserService {

    @Autowired
    private SessionFactory sessionFactory; // (1)

    public void registerUser(User user) {

        Session session = sessionFactory.openSession(); // (2)

        // lets open up a transaction. remember setAutocommit(false)!
        session.beginTransaction();

        // save == insert our objects
        session.save(user);

        // and commit it
        session.getTransaction().commit();

        // close the session == our jdbc connection
        session.close();
    }
}
```

1. This is a plain, old Hibernate SessionFactory, the entry-point for all Hibernate queries.
2. Manually managing sessions (read: database connections) and transactions with Hibernate's API.

There is one huge problem with the above code, however:

- Hibernate would not know about Spring's @Transactional annotation.
- Spring's @Transactional would not know anything about Hibernate's transaction.

But we'd actually *love* for Spring and Hibernate to integrate seamlessly, meaning that they know about each others' transactions.

In plain code:

```
@Service
public class UserService {

    @Autowired
    private SessionFactory sessionFactory; // (1)

    @Transactional
    public void registerUser(User user) {
        sessionFactory.getCurrentSession().save(user); // (2)
    }

}
```

1. The same SessionFactory as before
2. But no more manual state management. Instead, getCurrentSession() and @Transactional are *in sync*.

How to get there?

Using the HibernateTransactionManager

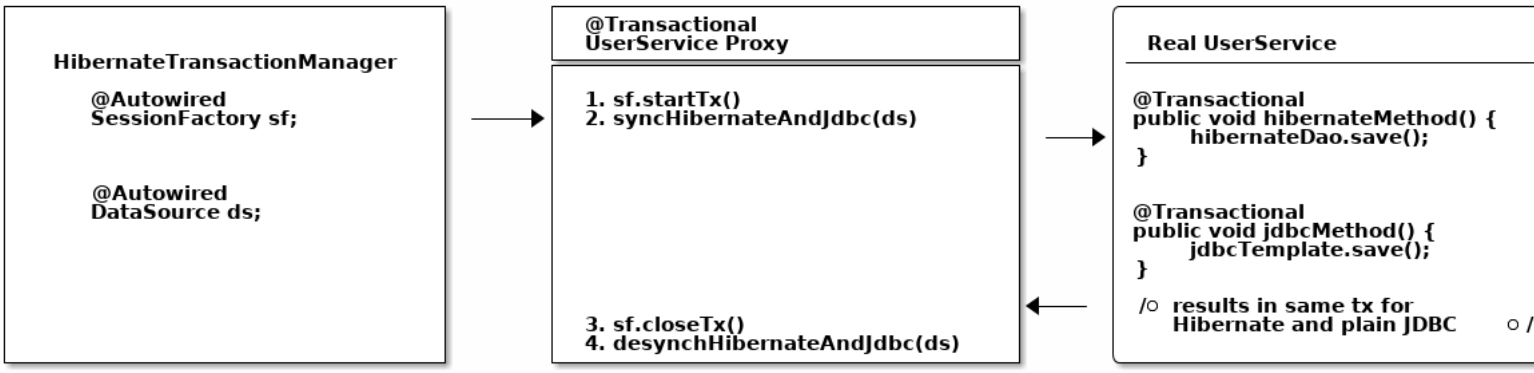
There is a very simple fix for this integration problem:

Instead of using a DataSourcePlatformTransactionManager in your Spring configuration, you will be using a HibernateTransactionManager (if using plain Hibernate) or JpaTransactionManager (if using Hibernate through JPA).

The specialized HibernateTransactionManager will make sure to:

1. Manage transactions through Hibernate, i.e. the SessionFactory.
2. Be smart enough to allow Spring to use that very same transaction in non-Hibernate, i.e. @Transactional Spring code.

As always, a picture might be simpler to understand (though note, the flow between the proxy and real service is only conceptually right and oversimplified).



That is, in a nutshell, how you integrate Spring and Hibernate.

For other integrations or a more in-depth understanding, it helps to have a quick look at all possible PlatformTransactionManager implementations that Spring offers.

Fin

By now, you should have a pretty good overview of how transaction management works with the Spring framework and how it also applies to other Spring libraries like Spring Boot or Spring WebMVC. The biggest takeaway should be, that it does not matter which framework you are using in the end, it is all about the JDBC basics.

Get them right (Remember: *getConnection(). setAutocommit(false). commit().*) and you will have a much easier understanding of what happens later on in your complex, enterprise application.

Thanks for reading.

Acknowledgements

Thanks to Andreas Eisele for feedback on the early versions of this guide. Thanks to Ben Horsfield for coming up with much-needed Javascript snippets to enhance this guide.