Spring transactional Management

**Error**

* It is classified as an unchecked type.
* It belongs to the class ‘java.lang.error’.
* It can’t be recovered from.
* It can't occur at compile time.
* Examples of errors include −
  + ‘OutOfMemoryError’
  + ‘IOError’

**Exception**

* It can be classified into unchecked and checked exceptions.
* It belongs to the class ‘java.lang.Exception’.
* It can be recovered from.
* It can occur at runtime as well as compile time.
* Examples of exceptions include −
  + NullPointerException
  + SqlException

1. @Transactional

* Wrap a method
* Can be called at Interface ,call ,method levels
* Annotation on private and protected method , Spring will ignore it.
* Not advisable use @Trasnactional at Interface level.unless you want to override the transaction setting of interface.
* Set propagation, isolation ,timeout and readonly,rollback for our transactions.
* Programmatic Transaction Management
* Declarative Transaction Management

1. Physical and logical transactions
2. @Transactional and JPA Hibernate Transaction
3. @Transactional with Spring boot or Spring MVC Integration
4. Rollbacks , Proxies,Common Pitfalls, and much more

JDBC Transactions

* Datasource.getConnection
* setAutocommit to false
* commit
* in case of any exception – Connection.rollback();

HikariCP – Connection pool libraries

* toggle the autocommit mode automatically for us depending on the configuration

JDBC Isolation Levels and save points

|  |
| --- |
| @Transactional(propagation=TransactionDefinition.NESTED,  isolation=TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED) |

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);

savepoint support is dependent on your JDBC driver / database

We can use PlatformTransactionManager

-Low Level API to manage transactions manually

- @Transactional and Transaction Template uses this API to mange transactions internally

Springboot Transaction Mangement works

JDBC only offers setAutoCommit to false and spring offers many more to achieve convenient ways to archive the same.

Spring Programmatic Transaction Management

@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*

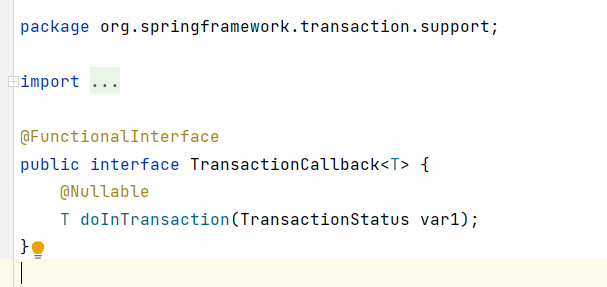
return id;

});

}

}

* Use Tranactional Callback to handle opening and closing database connetions
* No need of SQLException , spring converts to runtime Exceptions.



Spring declarative transaction Management

* @EnableTransactionManagement in spring configuration
* Specify transaction Manager in spring configuration

@Configuration

@EnableTransactionManagement

public class MySpringConfig {

@Bean

public PlatformTransactionManager txManager() {

return yourTxManager; *// more on that later*

}

}

* It initiates the UserService Bean and transactional Proxy of UserService

@Configuration

@EnableTransactionManagement

public static class MyAppConfig {

@Bean

public UserService userService() { // **(1)**

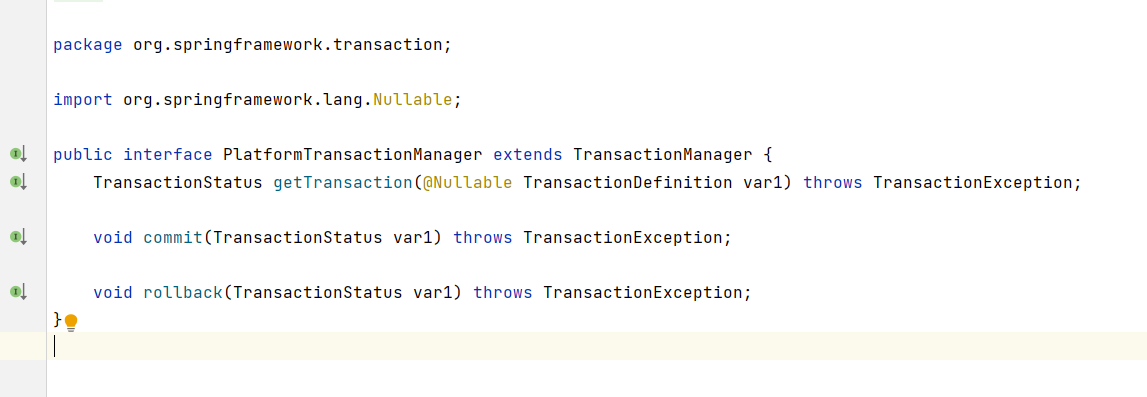
return new UserService();

}

}

Spring constructs a dynamic CGLib proxy of the UserService class to open and close database transactions. – Proxy wrapping your UserService

Proxy delegates the work to transaction manager



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);

}

}

}

This does the same thing as performed by JDBC

* @Transactional annotation creates a dynamic proxy of the bean
* Proxy has access to transaction manager for opening and closing of transaction and connections.
* Does the same as JDBC connection

Physical and Logical transactions

* Uses the same underlying , physical database connection.

@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() {

*// ...*

}

}

@Service

public class InvoiceService {

@Transactional(propagation = Propagation.REQUIRES\_NEW)

public void createPdf() {

*// ...*

}

}

Requires\_new - tells spring that createPDF t oexecute in own transaction without any dependency.

In this case 2 physical connection are established

Physical Transactions : Actual JDBC transactions

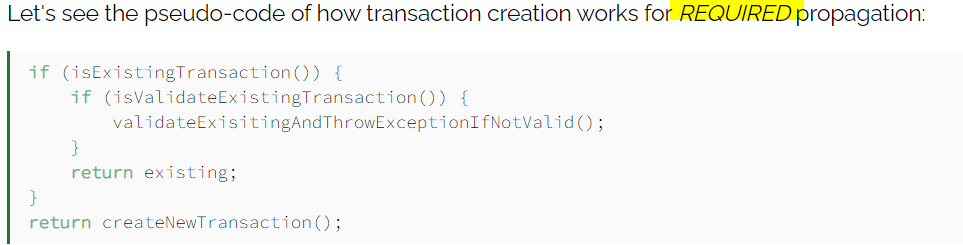
Logical Transaction : @Transactional annotated methods

Propagation Levels :

1. REQUIRED
2. SUPPORTS
3. MANDATORY
4. REQUIRES\_NEW
5. NOT\_SUPPORTED
6. NEVER
7. NESTED

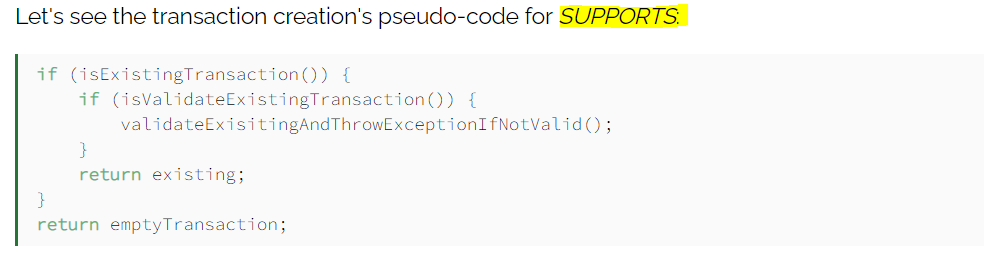
Required :

* Default propagation
* Checks for active connection,if not create a new connection.
* getConnection().setAutocommit(false).commit().



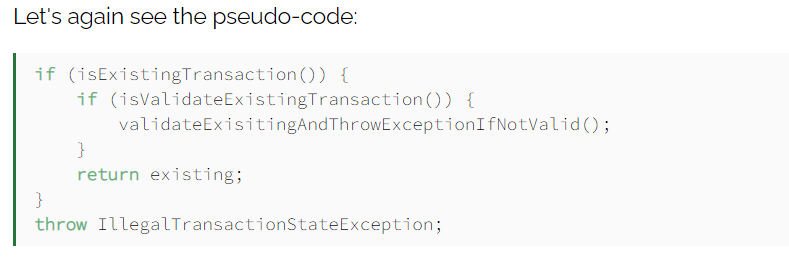
Supports :

* Nothing to do with JDBC
* It will be executed as non -transactional



Mandatory :

* If active transaction is t here it will be used ,else through exception.



Require\_new

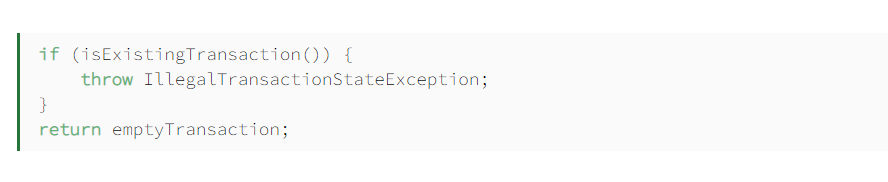
* Spring suspends the existing transaction and creates a new one .
* Completely owned connection.
* getConnection().setAutocommit(false).commit().

Not\_supported :

* First Spring suspends the current transaction and then business logic is executed without a transaction.

NEVER:

* Throw exception in case of active transaction.
* Noting to do with JDBC



Nested :

* Check for transaction exists,if so ,it marks a savepoint.
* If business logic execution throws exception then transaction rollbacks to savepoint.
* In case of no active connection. It acts as REQUIRED .
* Complicated.
* Connection.setsavepoint();

@Transactional Isolation Levels :

@Transactional(isolation = Isolation.REPEATABLE\_READ)

Is same as used in JDBC

connection.setTransactionIsolation(Connection.TRANSACTION\_REPEATABLE\_READ);

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.

How Spring MVC and Spring boot handles the @Transactional ???

Transaction Isolation :

Isolation is one of the common ACIS Properties

Atomicity

Consistency

Isolation

Durability

Describes how changes applied by concurrent transactions are visible to each other.

* **Dirty Read:**

Read the uncommited change of concurrent transcation

**Non repeatable read :**

Get different value on re-read of a row if concurrent transaction updates the same row and commits

**Phantom read :**

Get different rows after re-execution of range query,if another transaction adds or remove some rows in the range and commits

DEFAULT

READ\_UNCOMMITED

* + Lowest Isolation level
  + Gets different results on re-read, re-execution and uncommited data
  + Postgres and Oracle do not support READ\_UNCOMMITED and falls back to READ\_COMMITED

READ\_COMMMITED

* + Second level of Isolation
  + Prevents dirty reads
  + On requerying the data may change if the transaction commits its changes.
  + READ\_COMMITED is default level for Postgres ,Oracle and SQL Server .

REPEATABLE\_READ

* + Third level of Isolation
  + Prevents dirty and non-repeatable reads
  + Lowest level to prevent lost update
  + Lost update occurs when 2 or more concurrent transaction read and update the same row.
  + Doesn’t allow simultaneous access to row at all.
  + Itsdefault level in MySQL and Oracle doesn’t support REPEATABLE READ

SERIALIZABLE

* + Highest level of isolation
  + Execute concurrent calls sequentially