Chapter 5 – Spring Data Access

Basic way to access a data base is using JDBC connection, but using this approach developer needs to managed their own connection and handle JDBC objects. JDBC is redundant usages, prone to error, poor exception handling and cumbersome to use.

Memory leak – object still utilizing memory long after its needed or accessible.

**Spring JDBC template**

Spring JDBC framework consist of four major packages, and a small config package that house the infrastructure bean & utility classes for embedded databases.

* **Core**: Spring JDBCTeamplate class of the core package hides major boilerplate code and unburden developers from managing connections.
* **Data source**: The Datasource package contains the utility classes for datasource management
* **Object**: Object package contains the RDBMS quries , update , store procedures as thread-safe , reusable objects.
* **Support**: provides feature for SQLException translation functionality and some utility classes.

JDBC template is designed over template design pattern which is a behavior design pattern category.

Every time a JBDC template method is called, a new connection is automatically opened and close one the execution is complete. JDBC template is perfect for small project where one needs to get rid of connection management using JDBC connectors.

JDBCTemplate method queryForObject was change to specific native dataTypes like queryForInt, queryForLong,QueryForString

JdbcTemplate method queryForMap method return pairs of columns [column\_name(string), column\_value(object)]

The RowMapper object is stateless. RowMapper is a functional interface, one can implement a lambda function instead, there is no harm in defining one row-mapper per interface. RowMapper is not required if ORM products are used .

|  |
| --- |
| RowMapper<Person> personRowMapper = (rs,rowInt) -> {  return  Person.Builder  .setPersonName(rs.getString(“PERSON-NAME”))  .setPersonAge(rs.getInt(“AGE”)).build();  } |

**Things to know about JdbcTemplate**

* JdbcTemplate works with queries that specify parameters using the '?' placeholder.
* Use queryForObject when it is expected that execution of the query will return a single result.
* Use RowMapper<T> when each row of the ResultSet maps to a domain object.
* Use RowCallbackHandler when no value should be returned.
* Use ResultSetExtractor<T> when multiple rows, or multiple records from different tables returned in a ResultSet map to a single object.
* DDL – Data Definition Language are database operations that manipulate database objects, such as tables, views, cursors, and so forth.
* DML – Data Manipulation Language, the SELECT, INSERT, UPDATE, and DELETE commands are database statements to create, update, or delete data from existing tables.

JdbcTemplate can execute both DML and DDL statements. However, its not advisable to use jdbcTemplate for running DDL as there is a risk of sqlInjection.

jdbcTemplate.execute(“CREATE TABLE DUMMY\_TABLE”).

NOTE here table name cannot be provided a parameter as , its expected that parameter should be part of an expression whereas the table name is not part of any expression – if any one attempt to do so, it will throw JdbcSQLSyntaxErrorException

jdbcTemplate.execute(“select \* from DUMMY\_TABLE”,Integer.class);

**Exception handling in accessing data.**

NOTE: the checked exception is the once that needs to handle within the code or throws up to call method hierarchy, otherwise it will throw a compile time error. Whereas, the unchecked exception is once which are not checked by the compiler – even if they are NOT handle will not throw any error.

In pure JDBC same type of checked excetion java.sql.SQLException is thrown, which needs to be handle or thrown throw out the calling method hierarchy making in tightly coupled. In case of spring , all database exception are unchecked exception and are extend from RuntimeException class. Developer need not to handle the exception making it more loosely coupled.

Spring data base transaction are of three major types

* Exception those are consider as **non-transient** exception – these exceptions are not recoverable unless originating cause is fixed. The are extend from org.springframework.dao.

NonTransientDataAccessException abstract class.

* **Recoverable exception** – these exception can be recovered if certain recovery steps are performed extend from org.springframework.dao.RecoverableDataAccessException
* The third type is **transient** exception – this are those exception which can be recovered jut by retrying. These exceptions are extended from org.springframework.dao.TransientDataAccessException abstract class.

Apart from these three major exception categories there is another category which is cause due to wrong initialization of database or failure of initialization scripts.

**Data base transactions.**

For managing a database transaction, a transactions environment is required, spring usages an infrastructure bean called as transaction manager to manage the transactions – when moving from one environment to another only these bean needs to be change. There are basically four flavor of transaction manager environment available.

* **JDBC Spring environment:** Type org.springframework.jdbc.datasource.

DataSourceTransactionManage.

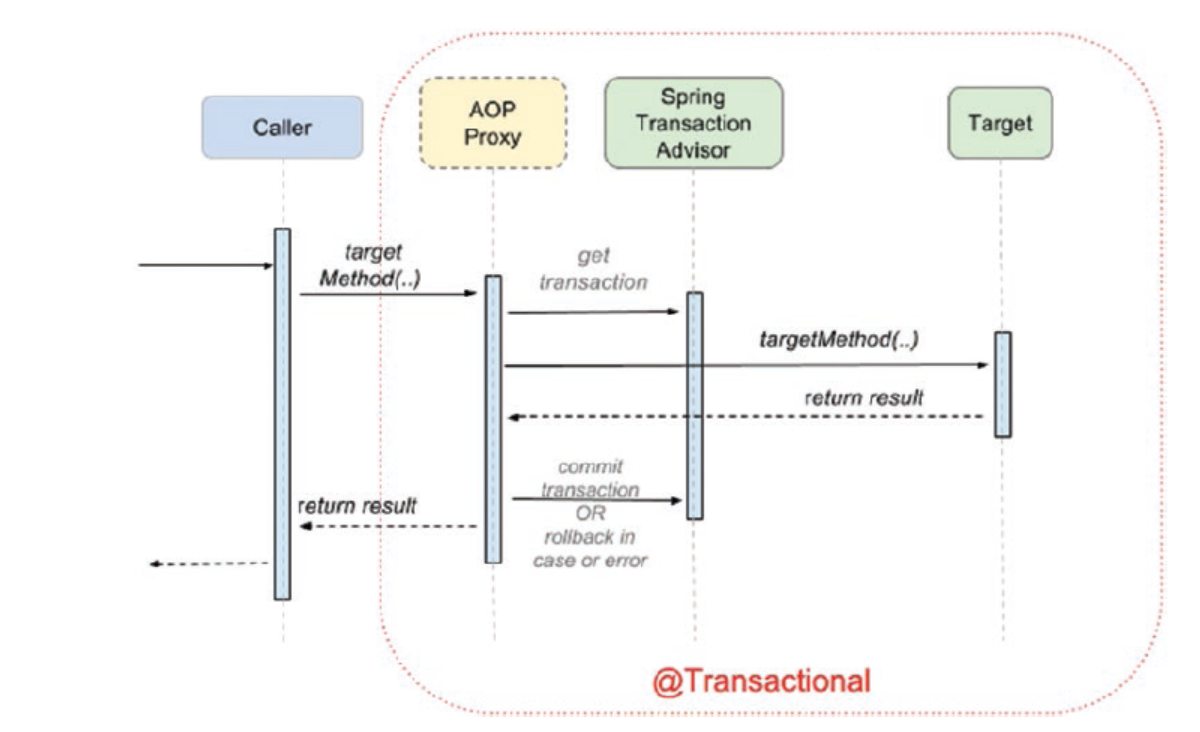
* **Hibernate Spring environment:** Type org.springframework.orm.

hibernate5.HibernateTransactionManager, a Spring-specific implementation that uses a hibernate session object created by an infrastructure bean of type org.springframework.orm.hibernate5.LocalSessionFactoryBean to manage entities in a transactional context.

* **JPA environment**: a bean of type org.springframework.orm.jpa.JpaTransactionManager, a Spring-specific implementation that uses an entity manager object created by an infrastructure bean of type org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean to manage entities in a transactional context.
* **Enterprise JTA environment**: setup requires an application server that will configure and provide a datasource bean using JNDI. JNDI works with other technologies on the Java platform, such as the Enterprise Edition (Java EE), to organize and locate components in a distributed computing environment. Spring loads a bean of type extending org.springframework.transaction. jta.JtaTransactionManager specific to the application server used. This transaction manager is appropriate for handling distributed transactions, which are transactions that span multiple resources, and for controlling transactions on application server resources.

In spring the transaction is handled through AOP as it’s a cross cutting concerns, AOP framework provide an around advice wrapping the original bean methods which executed in a transaction, retrieving or opening a transaction before execution and committing

it afterward is necessary.



The AOP proxies use two infrastructure beans for this: an org.springframework.transaction.interceptor.TransactionInterceptor in conjunction with an implementation of org.springframework.transaction.PlatformTransactionManager. Spring provides a flexible and powerful abstraction layer for transaction management support.

In Spring any method that is annotated with @Transactions is executed as single unit of transaction. For AOP annotation to work, the method should be public otherwise the transaction cannot be managed.

The simplest way to configure an transaction manager is to create a bean of type PlatformTransactionManager and add an class level annotation @EnableTransactionManagement.

The @EnableTransactionManagement is more flexible; it looks for a bean of any type

that implements the org.springframework.transaction.PlatformTransactionManager,

so the name is not important. In case the default transaction manager bean must

be established without a doubt, this can be done by making the configuration class.

In case of bigger application where it needs more than one datasource there needs to have multiple transaction manger , if in the @Transaction annotation the transaction manger to be used is not defined then the spring application would start successfully but fails when the transactional method is called , as spring wouldn’t check transaction manager beans until transactional method is called. When it checks it would find two bean of Type PlatformTransactionManager, and spring cannot decide on its own which one to us. This problem can be overcome by specifying transaction manager to be use in @Transactional(transactionManager = "simpleManager", readOnly = true). Alternatively one can also used @Primary to ensure spring chose the right transaction manager OR make sure the configuration class implements TransactionManagementConfigurer interface which has a single method annotationDrivenTransactionManager() that specifies default transaction.

The flowing are the transaction propagation behaviors, one of these behaviors can be selected while executing an existing transaction OR creating a new transaction.

| **Propagation** | **Description** |
| --- | --- |
| REQUIRED: | An existing transaction will be used or a new  one will be created to execute the method annotated with @  Transactional(propagation = Propagation.REQUIRED). |
| REQUIRES\_NEW | A new transaction is created to execute the  method annotated with @Transactional(propagation =  Propagation.REQUIRES\_NEW). If a current transaction exists, it will be suspended. |
| NESTED: | An existing nested transaction executes the method  annotated with @Transactional(propagation = Propagation. NESTED). If no such transaction exists, it will be created. This approach is quite similar to REQUIRED, so if the datasource supports I think it should be mandatory to use it because it reuses existent resources. |
| MANDATORY: | An existing transaction must be used to execute  the method annotated with @Transactional(propagation =  MANDATORY). If there is no transaction to be used, an exception will be thrown. |
| NEVER: | Methods annotated with @Transactional(propagation =  Propagation.NEVER must not be executed within a transaction. If  a transaction exists, an exception will be thrown. |
| NOT\_SUPPORTED | No transaction executes the method annotated  with @Transactional(propagation = Propagation.NOT\_  SUPPORTED). If a transaction exists, it will be suspended. |
| SUPPORTS: | An existing transaction executes the method annotated  with @Transactional(propagation = Propagation.SUPPORTS).  If no transaction exists, the method will be executed anyway,without a transactional context. |

In general, the transactions should be isolated, one transaction should NOT access data from other transactions, however in DBMS there are four level of isolation provided, spring supports five different level of isolations.

|  |  |
| --- | --- |
| **Isolation** | **Description** |
| DEFAULT | The default isolation level of the DBMS. |
| READ\_UNCOMMITED | Data changed by a transaction can be read by a different transaction while the first one is not yet committed, also known as *dirty reads*. Dirty reads are possible at this isolation level. |
| READ\_COMMITTED | Dirty reads are not possible when a transaction is used with this isolation level. This is the default strategy for most databases. But a different phenomenon could happen here: *non-repeatable read*: when the same query is executed multiple times, different results might be obtained. (For example, a person is extracted repeatedly within the same transaction. In parallel, a different transaction edits the person and commits. If the first transaction has this isolation level, it will return the person with the new properties after the second transaction is committed.) |
| REPEATABLE\_READ | This level of isolation does not allow  dirty reads, and repeatedly querying a table row in the same transaction will always return the same result, even if a different transaction has changed the data and committed while the reading occurs. The process of reading the same row multiple times in the context of a transaction and always getting the same  result is called *repeatable read*. But, at this level, *phantom reads* are still possible. A *phantom read* happens when in the course of a transaction, the execution of identical queries leads to different result sets returned. |
| SERIALIZABLE | This is the most restrictive isolation level, since transaction are executed in a serialized way. So no dirty reads, no repeatable reads, and no phantom reads are possible. |

**Timeout** – this is one of the attributes of the @Transation annotation, default transaction time out is set by the value set within the transaction Manager object. This value can be override by setting a timeout value as attribute to the @Transaction annotation. Default value for the timeout attribute is set to 1 which means timeout not supported, other integer value can be set equivalent to the timeout in millisecond.

**rollbackFor attributes** – this @Transaction annotation attribute defines one or more exception or Throwable subclass , by default transaction is ONLY rollback when RuntimeException is thrown.

**noRollbackFor attribute** – this @Tansaction annotation attribute defines one or more exception for Throwable subclasses, with this one can also define runtime Exception as non rollback exception.

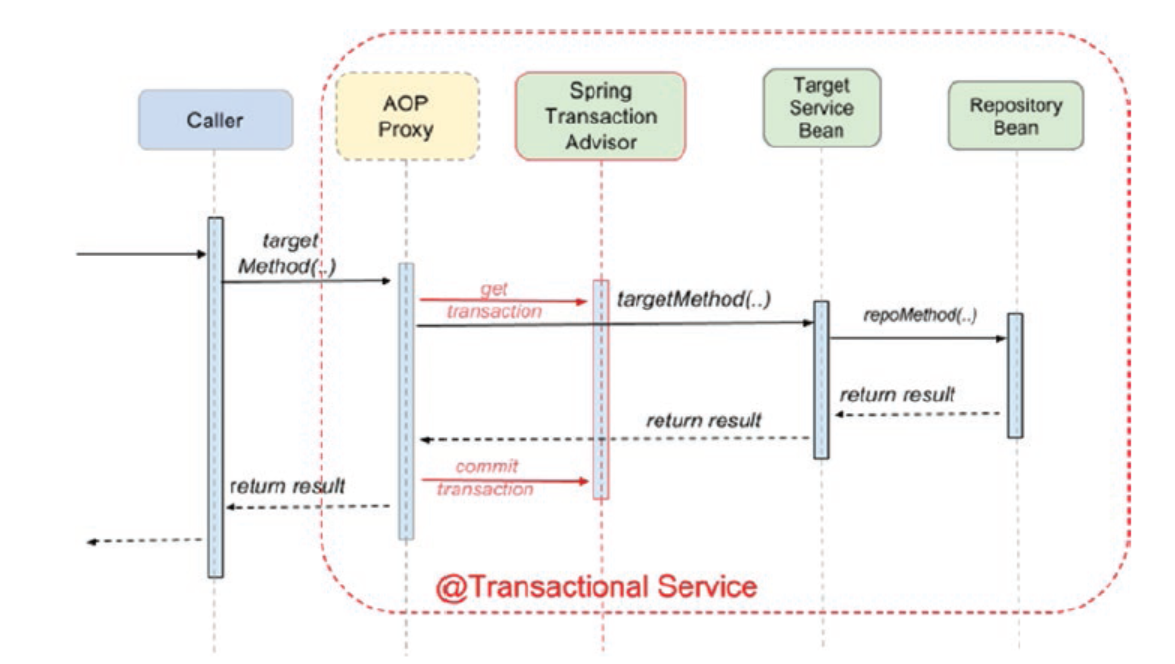
**NOTE: @Transcation can be used at the class level too, in that case all the methods defined in the class will inherited the same transactional behavior.**

Transaction is an cross cutting concerns and its been implemented in spring using AOP, the default behavior is to create an interface based proxy this behavior can be change by setting property proxyTargetClass = true of @EnableTransactionManager annotation.

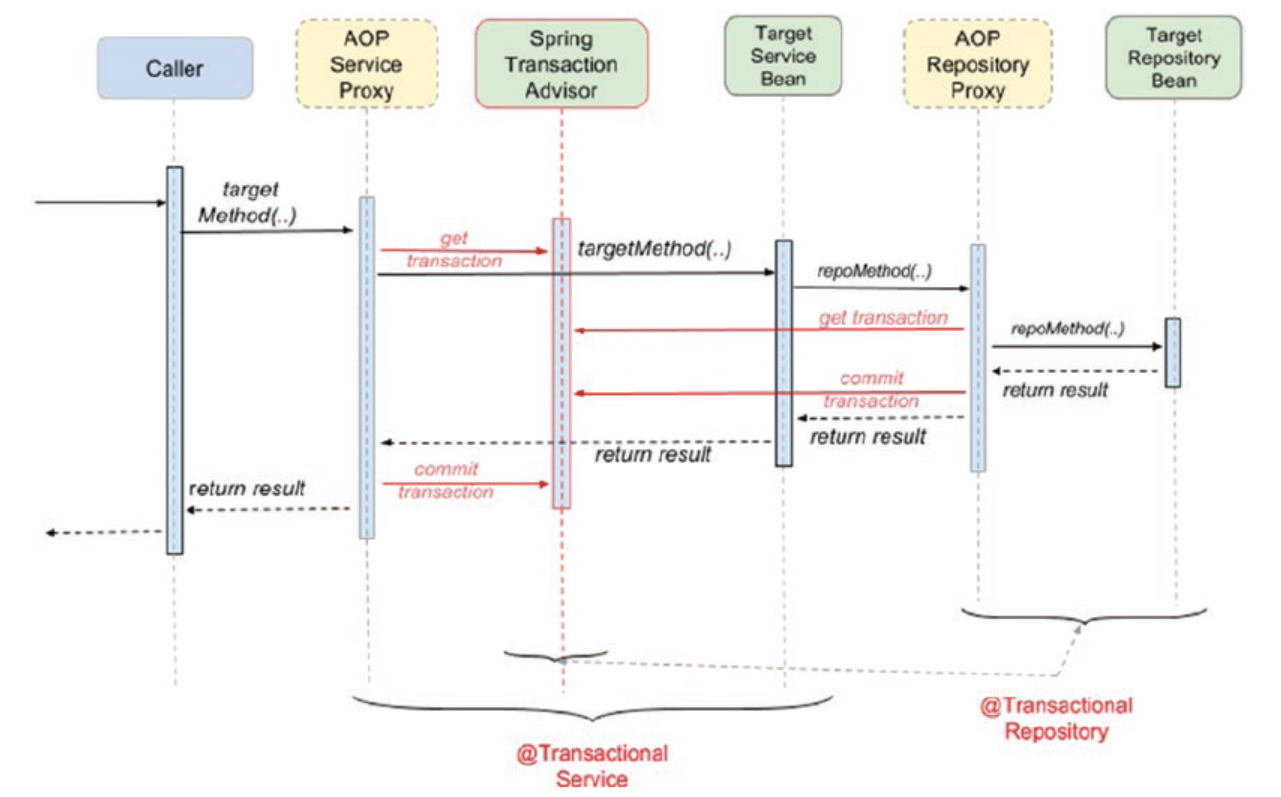
Its important that @Transaction annotation is to be applied at the public method level. (AOP cannot.

If @Transaction needs to be put for Service classes OR for the repository classes?

Spring usages decorative design pattern to wrap a target Bean – when a Service method is annotated with the @Transaction it creates a wrapper around the service target class object where service class method is executed with the accrued transaction context. If there are multiple repository methods are called within the same service class target method then all the methods are called within the same transaction context. Setting the @Transaction annotation only to the repository alone will not be suffices if the service method calls methods from different repository classes within some method, then they can’t be executed within a same transaction context.



Setting @Transaction annotation to both service and repository classes is redundant as the APO would be creating two wrapper one for service and other for repository classes and its needs to propagate the transaction from service to repository.



NOTE Use @Transactional in the service layer or the DAO/repository layer, but not both. The service layer is the usual choice, because service methods call multiple repository methods that need to be executed in the same transaction. The only reason to make your repositories transactional is if you do not need a service layer at all, which is usually the case for small educational applications.

Spring also provide way to handle the transaction proamativally where developer can manage their transactions – for that spring does provide TransactionTemplate , but the developer handle the transactions on their own.

|  |
| --- |
| public class ProgramaticDetectiveService implements DetectiveService  {  private DetectiveRepo detectiveRepo;  private TransactionTemplate txTemplate;  public ProgramaticDetectiveService(DetectiveRepo detectiveRepo,  PlatformTransactionManager transactionManager) {  this.detectiveRepo = detectiveRepo;  this.txTemplate = new TransactionTemplate(transactionManager);  }  @Override  public Optional<Detective> findById(Long id) {  return txTemplate.execute(status -> {  Optional<Detective> opt = null;  try {  opt = detectiveRepo.findById(id);  } catch (Exception e) {  status.setRollbackOnly();  }  return opt;  });  }  } |

State.SetRollbackOnly() method needs to be called when the transactions need to be rollbacked.

NOTE: Distributed transactions: The distributed transaction are those transactions that span across multiple environments – to manage this type of transactions one needs to use JTA java transaction API and specific XA drivers.

**Hibernate ORM (Object Relational Mapping) framework**

There are number of ORM projects available like Apache OpenJPA , Hibernate , EclipesLink etc. Hibernate ORM is an object relational mapping framework that provides support for mapping an object-oriented domain model to a relational database.

**Hibernate Properties**

* Hibernate.Dialect, this shou be matching with the data base which is currently been used.
* Hibernate.hbm2ddl, this represents what application should do when it starts up the following are the option available
  + Create-only
  + Drop
  + Create
  + Create-drop
  + Validate
  + Update
* hibernate.format\_sql Boolean value, if set to true generated sql statements will be make human readable on the console.
* Hibernate.show\_sql Boolean value, if this values to set as true then the generated sql will be displayed on the console.
* Hibernate.use\_sql\_comments Boolean value, if this value is set to true than hibernate will insert comments to generated SQL for developer to understand better.

Benefit of using Hibernate ORM

Spring Data JPA

Spring picks up the type interface and implements them at the runtime to create a bean.

There are various repositories available,

NOTE : if spring-boot-starter-data-jpa module is on the classpath, there is no need to add @EnableJpaRepositories or @EnableTransactionManagement on a configuration class.