**SPRING DATA MANAGEMENT**

**What is the difference between checked and unchecked exceptions?**

***Checked exceptions:*** Java compiler requires to handle or declare. E.g., Exception

***Unchecked exceptions:*** compiler not required to handle or declare. E.g., RuntimeException.

Checked exceptions are invalid conditions.

Unchecked exceptions are usually defects.

**Why does Spring prefer unchecked exceptions?**

Checked exceptions rewires handling, result in cluttered code and unnecessary coupling.

Spring prefers unchecked exception because this way it gives the developer possibility to choose to handle them or not.

**What is the data access exception hierarchy?**

Each data access technology has its own exception types, such as

* SQLException for direct JDBC access,
* HibernateException used by native Hibernate, or
* EntityException used by JPA

**What Spring does is to handle technology-specific exceptions and translate them into its own exception hierarchy?**

***Spring’s DataAccessException -***

* It is an abstract class.
* It is the root exception.
* Its sub-classes are unchecked exceptions.

Spring data access exception family has three main branches:

***1) org.springframework.dao.NonTransientDataAccessException***

* Non-transient exceptions.
* which means that retrying the operation will fail unless the originating cause is fixed.
* The most obvious example here is searching for an object that does not exist.

***2) springframework.dao.TransientDataAccessException.***

* Transient exception.
* which means that retrying the operation might succeed without any intervention.
* These are concurrency or latency exceptions.

For example, when the database becomes unavailable because of a bad network connection in the middle of the execution of a query, an exception of type QueryTimeoutException is thrown. The developer can treat this exception by retrying the query.

***3) RecoverableDataAccessException***

When a previously failed operation might succeed if some recovery steps are performed, usually closing the current connection and using a new one.

E.g., a temporary network hiccup

**What is difference between HQL and JPQL?**

HQL (Hibernate Query Language) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. This is main difference between hql vs sql. HQL is a superset of the JPQL, the Java Persistence Query Language.

**What is the difference between JPQL and SQL?**

JPA (Java Persistence API) has its own Query language called JPQL. JPQL is very similar to SQL, with one major difference being that JPQL works with entities as defined in the application whereas SQL works with table and column names as defined in the database.

**What is HSQLDB?**

HSQLDB (HyperSQL DataBase) is a relational database management system written in Java. It has a JDBC driver and supports a large subset of SQL-92, SQL:2008, and SQL:2011 standards. It offers a fast, small database engine which offers both in-memory and disk-based tables. Both embedded and server modes are available.

**How do you configure a DataSource in Spring?**

Spring obtains a connection to the database through a DataSource.

DataSource VS Connection: DataSource provides and manages Connections.

Spring offers several options for configuring data-source beans, including:

***1) Data sources that are looked up by JNDI -***

The Java Naming and Directory Interface (JNDI) is a Java API for a directory service that allows Java software clients to discover and look up data and resources (in the form of Java objects) via a name.

The JNDI data source accesses a database connection that is pre-defined and configured in the application server and published as a JNDI resource or service. Instead of specifying a driver and database as we do with JDBC data sources, we only need to specify the JNDI resource name in our application server.

JNDI data sources can be managed completely external to the application, allowing the application to ask for a data source when it’s ready to access the database. Moreover, data sources managed in an application server are often pooled for greater performance and can be hot-swapped by system administrators.

JNDI comes in rescue when you have to move an application between environments: development -> integration -> test -> production.

If you configure each application server to use the same JNDI name, you can have different databases in each environment but you need not to change your code. You just need to drop the deployable WAR file in the new environment.

***@Bean***

***public JndiObjectFactoryBean dataSource() {***

***JndiObjectFactoryBean jndiObjectFB = new JndiObjectFactoryBean();***

***jndiObjectFB.setJndiName("jdbc/demodb");***

***jndiObjectFB.setResourceRef(true);***

***jndiObjectFB.setProxyInterface(javax.sql.DataSource.class);***

***return jndiObjectFB;***

***}***

In SpringBoot, it can be set in properties file as -

spring.datasource.jndi-name=java:jdbc/customers

***2) Using a pooled data source -***

Connection pooling is a well-known data access pattern, whose main purpose is to reduce the overhead involved in performing database connections and read/write database operations. In a nutshell, a connection pool is, at the most basic level, a database connection cache implementation, which can be configured to suit specific requirements.

It becomes evident that database connections are fairly expensive operations, and as such, should be reduced to a minimum in every possible use case (in edge cases, just avoided). By just simply implementing a database connection container, which allows us to reuse a number of existing connections, we can effectively save the cost of performing a huge number of expensive database trips, hence boosting the overall performance of our database-driven applications.

Spring doesn’t provide a pooled data source. However, there are other JDBC Connection Pooling Frameworks available like Apache Commons DBCP, HikariCP and C3PO.

***a) Apache Commons DBCP -***

***public class DBCPDataSource {***

***private static BasicDataSource ds = new BasicDataSource();***

***static {***

***ds.setUrl("jdbc:h2:mem:test");***

***ds.setUsername("user");***

***ds.setPassword("password");***

***ds.setMinIdle(5);***

***ds.setMaxIdle(10);***

***ds.setMaxOpenPreparedStatements(100);***

***}***

***public static Connection getConnection() throws SQLException {***

***return ds.getConnection();***

***}***

***private DBCPDataSource(){ }***

***}***

***Get a pooled connection with the DBCPDataSource class:***

Connection con = DBCPDataSource.getConnection();

***b) HikariCP -***

***public class HikariCPDataSource {***

***private static HikariConfig config = new HikariConfig();***

***private static HikariDataSource ds;***

***static {***

***config.setJdbcUrl("jdbc:h2:mem:test");***

***config.setUsername("user");***

***config.setPassword("password");***

***config.addDataSourceProperty("cachePrepStmts", "true");***

***config.addDataSourceProperty("prepStmtCacheSize", "250");***

***config.addDataSourceProperty("prepStmtCacheSqlLimit", "2048");***

***ds = new HikariDataSource(config);***

***}***

***public static Connection getConnection() throws SQLException {***

***return ds.getConnection();***

***}***

***private HikariCPDataSource(){}***

***}***

***Get a pooled connection with the HikariCPDataSource class:***

Connection con = HikariCPDataSource.getConnection();

***c) C3PO -***

***public class C3poDataSource {***

***private static ComboPooledDataSource cpds = new ComboPooledDataSource();***

***static {***

***try {***

***cpds.setDriverClass("org.h2.Driver");***

***cpds.setJdbcUrl("jdbc:h2:mem:test");***

***cpds.setUser("user");***

***cpds.setPassword("password");***

***} catch (PropertyVetoException e) {***

***// handle the exception***

***}***

***}***

***public static Connection getConnection() throws SQLException {***

***return cpds.getConnection();***

***}***

***private C3poDataSource(){}***

***}***

***Get a pooled connection with theC3poDataSource class :***

Connection con = C3poDataSource.getConnection();

In summary, It's possible to create JDBC connection pooling with the DriverManager class (custom implementation) or with Datasource implementations (like Apache Commons DBCP, HikariCP and C3PO).

The most relevant point to stress here is that once the pool is created, connections are fetched from the pool, so there's no need to create new ones. Furthermore, when a connection is released, it's actually returned back to the pool, so other clients can reuse it.

***In SpringBoot, configure the datasource (hikari) in properties file -***

In Spring Boot 2, Hikari is the default DataSource implementation.

***This is what's changed from Spring Boot 1.x:***

1) The dependency to Hikari is now automatically included in spring-boot-starter-data-jpa

2) The discovery algorithm that automatically determines a DataSource implementation now prefers Hikari over TomcatJDBC.

Thus, we have nothing to do if we want to use Hikari in an application based on Spring Boot 2.x.

spring.datasource.hikari.maximum-pool-size=5

spring.datasource.hikari.minimum-idle=2

spring.datasource.hikari.leak-detection-threshold=20000

***3) Using JDBC driver-based data sources -***

The simplest data source you can configure in Spring is one that’s defined through a JDBC driver. Spring offers three such data-source classes to choose from:

***a) DriverManagerDataSource -***

The simplest implementation of a DataSource.

Returns a new connection every time.

Connections are not pooled.

It performs poorly when multiple requests for a connection are made.

Unsuitable for anything other than testing.

Capable of supporting multiple threads, they incur a performance cost for creating a new connection each time a connection is requested.

***@Bean***

***public DataSource dataSource() {***

***DriverManagerDataSource ds = new DriverManagerDataSource();***

***ds.setDriverClassName("org.h2.Driver");***

***ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");***

***ds.setUsername("sa");***

***ds.setPassword("");***

***return ds;***

***}***

***b) SimpleDriverDataSource***

Similar as DriverManagerDataSource

Capable of supporting multiple threads, they incur a performance cost for creating a new connection each time a connection is requested.

Except that it works with the JDBC driver directly

***c) SingleConnectionDataSource***

Returns the same connection every time a connection is requested.

Wraps a single Connection that is not closed after each use

Obviously, this is not multi-threading capable.

It isn’t exactly a pooled data source, you can think of it as a data source with a pool of exactly one connection.

***4) Using an embedded data source -***

An embedded database runs as part of your application instead of as a separate database server that your application connects to.

It’s not very useful in production settings.

An embedded database is a perfect choice for development and testing purposes. That’s because it allows you to populate your database with test data that’s reset every time you restart your application or run your tests.

In SpringBoot, Spring Boot detects that you have the H2 database library in your application’s classpath, it will automatically configure an embedded H2 database.

With Java configuration, you can use EmbeddedDatabaseBuilder to construct the DataSource as below-

***@Bean***

***public DataSource dataSource() {***

***return new EmbeddedDatabaseBuilder()***

***.setType(EmbeddedDatabaseType.H2)***

***.addScript("classpath:schema.sql")***

***.addScript("classpath:test-data.sql")***

***.build();***

***}***

**How profiles can be used to select datasource at runtime?**

Using profiles, the datasource is chosen at runtime, based on which profile is active. For instance, an embedded database is created if and only if the 'dev' profile is active. Similarly, the DBCP BasicDataSource is created if and only if the 'qa' profile is active. And the data source is retrieved from JNDI if and only if the 'prod' profile is active.

**Which bean is very useful for development/test databases?**

The embedded DB/datasource is a perfect choice for development and testing purposes. It’s not very useful in production settings.

It runs as part of application instead of being a standalone element.

It allows you to populate your database with test data that’s reset every time you restart your application or run your tests.

Spring Boot can auto-configure embedded H2, HSQL, and Derby databases.

In SpringBoot, Spring Boot detects that you have the H2 database library in your application’s classpath, it will automatically configure an embedded H2 database.

Spring Boot automatically creates the schema of an embedded DataSource. This behavior can be customized by using the 'spring.datasource.initialization-mode' property.

For instance, if you want to always initialize the DataSource regardless of its type:

spring.datasource.initialization-mode=always

**What is the Template design pattern and what is the JDBC template?**

Template design pattern

A template method defines the skeleton of a process.

At certain points, the process delegates its work to a subclass to fill in some implementation-specific details. This is the variable part of the process.

Spring separates the fixed and variable parts of the data-access process into two distinct classes: templates and callbacks.

***Templates manage the fixed of data access -***

* Establishing connection
* Handling transactions
* Handling excetions
* Clean up and release resource

***Your custom data-access code is handled in callbacks -***

* Creating statements,
* Binding parameters, and
* Marshaling result sets

***JDBC template***

The Spring JdbcTemplate is the central class in the JDBC core package. It simplifies the use of JDBC and helps to avoid common errors. It executes core JDBC workflow, leaving application code to provide SQL and extract results. This class executes SQL queries or updates, initiating iteration over ResultSets and catching JDBC exceptions and translating them to the generic, more informative exception hierarchy defined in the org.springframework.dao package.

***Benefits are:***

* Simplification: reduces boilerplate code for operations
* creation and closing of connections
* executing statements and stored procedure calls
* iterating over the ResultSet and returning results
* Handle exceptions
* Translates Exception from different vendors, e.g., DataAccessException
* Avoids common mistake: release connections
* Allows customization, it’s template design pattern
* Thread safe

***Spring comes with three template classes to choose from:***

* JdbcTemplate
* NamedParameterJdbcTemplate
* SimpleJdbcTemplate (deprecated) - The JdbcTemplate and NamedParameterJdbcTemplate now provide all the functionality of the SimpleJdbcTemplate.

***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.

jdbcTemplate.query for multiple rows or list

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 in the ResultSet map to a single object.

Initialize JdbcTemplate The general practice is to initialize JdbcTemplate within the setDataSource method so that once the data source is injected by Spring, JdbcTemplate will also be initialized and ready for use.

Once configured, JdbcTemplate is thread-safe. That means you can also choose to initialize a single instance of JdbcTemplate in Spring’s configuration and have it injected into all DAO beans.

***NamedParameterJdbcTemplate -***

Template class with a basic set of JDBC operations, allowing the use of named parameters rather than traditional '?' placeholders. Under the hood, it substitutes the named parameters to JDBC “?” placeholder and delegates to the wrapped JDCTemplate to execute the queries.

With NamedParameterJdbcTemplate, developer do not have to remember sequence of placeholders which improves maintainability of code. It also allows for expanding a List of values to the appropriate number of placeholders.

Named parameters will make your code easier to read hence better maintainable.

**What is a callback? What are the three JdbcTemplate callback interfaces that can be used with queries? What is each used for?**

Spring separates the fixed and variable parts of the data-access process into two distinct classes: Templates and callbacks.

* Templates manage the fixed of data access
* controlling transactions,
* managing resources, and
* handling exceptions.
* your custom data-access code is handled in callbacks
* creating statements,
* binding parameters, and
* marshaling result sets

A callback is code or reference to a piece of code that is passed as an argument to a method that, at some point during the execution of the methods, will call the code passed as an argument.

Spring converts contents of a ResultSet into domain objects using a callback approach.

The three callback interfaces that can be used with queries to extract result data are:

1) ResultSetExtractor: processes multiple rows, extractData() returns an object.

2) RowCallbackHandler: processes row by row, processRow() is void. Use when no value should be returned.

3) Rowmapper: processes row by row, mapRow()returns an object. Converts the row into an object

RowCallbackHandler - An interface used by JdbcTemplate for processing rows of a ResultSet on a per-row basis

CallableStatementCreator - One of the three central callback interfaces used by the JdbcTemplate class. This interface creates a CallableStatement given a connection, provided by the JdbcTemplate class.

CallableStatementCallback<T> action - Implemented as callback action working on a JDBC CallableStatement.

PreparedStatementCreator - One of the two central callback interfaces used by the JdbcTemplate class. This interface creates a PreparedStatement given a connection, provided by the JdbcTemplate class.

PreparedStatementCallback<T> action - Implemented as callback action working on a JDBC PreparedStatement.

**Can you execute a plain SQL statement with the JDBC template?**

Yes. With following methods:

* batchUpdate()
* execute()
* query()
* queryForList()
* queryForObject()
* queryForRowSet()
* update()

***DML***

DML stands for Data Manipulation Language, the commands SELECT, INSERT, UPDATE, and DELETE are database statements used to create, update, or delete data from existing tables.

***DDL***

DDL stands for Data Definition Language, used to manipulate database objects: tables, views, cursors, etc. DDL database statements can be executed with JdbcTemplate using the execute method.

**When does the JDBC template acquire (and release) a connection, for every method called or once per template? Why?**

It depends on the JdbcTempate's DataSource. If you provided a connection pool, like Apache commons-DBCP, then DBCP will do its best to reuse Connections. If you used Spring JDBC's DriverManagerDataSource a new Connection will be created / closed on each JdbcTemplate.query call

**What is the difference between execute(String sql) and update(String sql) in JdbcTemplate?**

The method execute(String sql) returns void if a call of the method succeeds without errors. As for plain JDBC, it should/can be used to define database schema elements (DDL), for instance with CREATE TABLE... statements. By contrast, update(String sql) is typically used for DML statements which correspond to SQL INSERT/UPDATE/DELETE operations. In these cases, in which data are manipulated, from a programmer's perspective it is important to know how many rows have been added/changed/deleted by the respective DML operation. For this reason, the update(...) method returns a non negative int value to indicate the number of rows affected.

**How jdbcTemplate translates an exception from different vendors to DataAccessException?**

Spring comes with its own data exception hierarchy out of the box – with DataAccessException as the root exception – and it translates all underlying raw exceptions to it. And so we keep our sanity by not having to handle low-level persistence exceptions and benefit from the fact that Spring wraps the low-level exceptions in DataAccessException or one of its sub-classes.

Also, this keeps the exception handling mechanism independent of the underlying database we are using.

Besides, the default SQLErrorCodeSQLExceptionTranslator, we can also provide our own implementation of SQLExceptionTranslator.

Here's a quick example of a custom implementation, customizing the error message when there is a duplicate key violation, which results in error code 23505 when using H2:

***public class CustomSQLErrorCodeTranslator extends SQLErrorCodeSQLExceptionTranslator {***

***@Override***

***protected DataAccessException customTranslate***

***(String task, String sql, SQLException sqlException) {***

***if (sqlException.getErrorCode() == 23505) {***

***return new DuplicateKeyException(***

***"Custom Exception translator - Integrity constraint violation.", sqlException);***

***}***

***return null;***

***}***

***}***

To use this custom exception translator, we need to pass it to the JdbcTemplate by calling setExceptionTranslator() method:

***CustomSQLErrorCodeTranslator customSQLErrorCodeTranslator = new CustomSQLErrorCodeTranslator();***

***jdbcTemplate.setExceptionTranslator(customSQLErrorCodeTranslator);***

**How JDBC operations are performed using SimpleJdbc classes?**

SimpleJdbc classes provide an easy way to configure and execute SQL statements. These classes use database metadata to build basic queries. SimpleJdbcInsert and SimpleJdbcCall classes provide an easier way to execute insert and stored procedure calls.

SimpleJdbcInsert

The INSERT statement is generated based on the configuration of SimpleJdbcInsert and all we need is to provide the Table name, Column names and values.

SimpleJdbcInsert simpleJdbcInsert = new SimpleJdbcInsert(dataSource).withTableName("EMPLOYEE");

Next, let's provide the Column names and values, and execute the operation:

***public int addEmplyee(Employee emp) {***

***Map<String, Object> parameters = new HashMap<String, Object>();***

***parameters.put("ID", emp.getId());***

***parameters.put("FIRST\_NAME", emp.getFirstName());***

***parameters.put("LAST\_NAME", emp.getLastName());***

***parameters.put("ADDRESS", emp.getAddress());***

***return simpleJdbcInsert.execute(parameters);***

***}***

We can also pass in parameters by using the BeanPropertySqlParameterSource and MapSqlParameterSource.

***Stored Procedures with SimpleJdbcCall***

***SimpleJdbcCall simpleJdbcCall = new SimpleJdbcCall(dataSource)***

***.withProcedureName("READ\_EMPLOYEE");***

***public Employee getEmployeeUsingSimpleJdbcCall(int id) {***

***SqlParameterSource in = new MapSqlParameterSource().addValue("in\_id", id);***

***Map<String, Object> out = simpleJdbcCall.execute(in);***

***Employee emp = new Employee();***

***emp.setFirstName((String) out.get("FIRST\_NAME"));***

***emp.setLastName((String) out.get("LAST\_NAME"));***

***return emp;***

***}***

**How does the JdbcTemplate support generic queries? How does it return objects and lists/maps of objects?**

JdbcTemplate supports Java 5-enhanced syntax with generics and varargs. It uses object of the required type as argument or return parameter using generics and Object data type so that it can be type-casted to required object.

JdbcTemplate supports querying for any type of object assuming you supplied a RowMapper/ResultSetExtractor interface implementation defining the way database table should be mapped to some entity type.

JdbcTemplate catches JDBC exceptions and translates them to the generic, more informative exception hierarchy defined in the org.springframework.dao package. That way it keeps the exception handling mechanism independent of the underlying database.

It has many overloaded methods for querying the database, but mainly you can divide them in:

* query()
* queryForObject() if you are expecting only one object
* queryForMap() will return a map containing each column value as key(column name)/value(value itself) pairs.
* queryForList() a list of above if you’re expecting more results

**Transactions**

**What is a transaction? What is the difference between a local and a global transaction?**

A database transaction is a critical part of any application. A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either be completed entirely or take no effect at all. The management of the sequence of actions is known as transaction management.

Transactions are described as ACID, which stands for the following:

***1) Atomicity***: A transaction should be treated as a single unit of operation, which means that either the entire sequence of operations is completed, or it takes no effect at all.

***2) Consistency:*** Once a transaction is completed and committed, then your data and resources will be in a consistent state that conforms to business rules.

***3) Isolation:*** If many transactions are being processed with the same dataset at the same time, then each transaction should be isolated from others to prevent data corruption.

***4) Durability:*** Once a transaction has completed, the results of the transaction are written to persistent storage and cannot be erased from the database due to system failure.

A global transaction is an application server managed and spreads across many components/ technologies (means distributed transaction management); whilst a local transaction is a simple transaction that is about one single database.

Global transaction allows to span multiple transactional resources, typically relational databases and message queues whereas, local transaction is resource specific such as a transaction associated with a JDBC connection or single DB.

JEE offers JTA to deal with global transactions. @Transactional annotation in Spring manages global transaction whereas @Transactional annotation provided by JPA is capable of handling local transactions only.

**Is a transaction a cross cutting concern? How is it implemented by Spring?**

Yes, transaction management is a cross-cutting concern.

AOP is used to decorate beans with transactional behavior. This means that when we annotate classes or methods with @Transactional, a proxy bean will be created to provide the transactional behavior, and it is wrapped around the original bean in an Around advice that takes care of getting a transaction before calling the method and committing the transaction afterward.

AOP proxies use two infrastructure beans for this:

* TransactionInterceptor

An implementation of PlatformTransactionManager interface. E.g.,

* DataSourceTransactionManager
* HibernateTransactionManager
* JpaTransactionManager
* JtaTransactionManager
* WebLogicJtaTransactionManager

Under the hood: An internal infrastructure Spring-specific bean of type InfrastructureAdvisorAutoProxyCreator is registered and acts as a bean postprocessor that modifies the service and repository bean to add transaction-specific logic. Basically, this is the bean that creates the transactional AOP proxy.

When an exception is thrown from within the body of the transactional method, Spring checks the exception type in order to decide if the transaction will commit or rollback.

**How are you going to define a transaction in Spring?**

We can implement transaction in Spring in 2 ways --

***A) Declarative transaction management -***

Declarative transaction management is non-invasive.

***Steps in configuration -***

1) Marked a Spring Configuration class using the @Configuration.

2) Using Java Configuration, define a PlatformTransactionManager bean using @bean.

3) Enable transaction management by annotate the config file with @EnableTransactionManagement.

***@Configuration***

***@EnableTransactionManagement***

***public class TestDataConfig {***

***@Bean***

***public PlatformTransactionManager txManager(){***

***return new DataSourceTransactionManager(dataSource());***

***}***

***}***

4) Declare transactional methods using @Transactional.

***@Service***

***public class UserServiceImpl implements UserService {***

***@Transactional(propagation = Propagation.REQUIRED, readOnly= true)***

***@Override***

***public User findById(Long id) {***

***return userRepo.findById(id);***

***}***

***}***

***B) Programmatic transaction management-***

Spring Framework provides two ways of implementing Programmatic Transaction:

a) Using TransactionTemplate, which is recommended by Spring. It uses a callback approach.

can write a TransactionCallback implementation, run execute(..)

***public class SimpleService implements Service {***

***// single TransactionTemplate shared amongst all methods in this instance***

***private final TransactionTemplate transactionTemplate;***

***// use constructor-injection to supply the PlatformTransactionManager***

***public SimpleService(PlatformTransactionManager transactionManager) {***

***this.transactionTemplate = new TransactionTemplate(transactionManager);***

***}***

***public Object someServiceMethod() {***

***return transactionTemplate.execute(new TransactionCallback() {***

***// the code in this method executes in a transactional context***

***public Object doInTransaction(TransactionStatus status) {***

***updateOperation1();***

***return resultOfUpdateOperation2();***

***}***

***});***

***}***

***}***

If there is no return value, use TransactionCallbackWithoutResult

***transactionTemplate.execute(new TransactionCallbackWithoutResult() {***

***protected void doInTransactionWithoutResult(TransactionStatus status) {***

***updateOperation1();***

***updateOperation2();***

***}***

***});***

Code within the callback can roll the transaction back by calling the setRollbackOnly() method on the supplied TransactionStatus object

***transactionTemplate.execute(new TransactionCallbackWithoutResult() {***

***protected void doInTransactionWithoutResult(TransactionStatus status) {***

***try {***

***updateOperation1();***

***updateOperation2();***

***} catch ( SomeBusinessException ex) {***

***status.setRollbackOnly();***

***}***

***}***

***});***

b) Using PlatformTransactionManager directly, which is low level.

First, pass the implementation of the PlatformTransactionManager (txManager) you use to your bean through a bean reference.

Then, by using the TransactionDefinition(def) and TransactionStatus(status) objects, you can initiate transactions, roll back, and commit.

***DefaultTransactionDefinition def = new DefaultTransactionDefinition();***

***// explicitly setting the transaction name is something that can be done only programmatically***

***def.setName("SomeTxName");***

***def.setPropagationBehavior(TransactionDefinition.PROPAGATION\_REQUIRED);***

***TransactionStatus status = txManager.getTransaction(def);***

***try {***

***// execute your business logic here***

***} catch (MyException ex) {***

***txManager.rollback(status);***

***throw ex;***

***}***

***txManager.commit(status);***

It is possible to use both declarative and programmatic transaction models simultaneously.

**What does @Transactional do?**

Describes a transaction attribute on an individual method, an interface or on a class.

At the class level, this annotation applies as a default to all methods of the declaring class and its subclasses. Note that it does not apply to ancestor classes up the class hierarchy; methods need to be locally redeclared in order to participate in a subclass-level annotation.

Rather than using XML AOP for matching the methods that should be transactional you can add this (@Transactional) annotation.

@Transactional Settings/attributes

The transactionManager attribute value defines the transaction manager used to manage the transaction in the context of which the annotated method is executed. May be used to determine the target transaction manager, matching the qualifier value (or the bean name) of a specific PlatformTransactionManager bean definition.

The readOnly attribute should be used for transactions that involve operations that do not modify the database (example: searching, counting records). Default FALSE. This just serves as a hint for the actual transaction subsystem; it will not necessarily cause failure of write access attempts. A transaction manager which cannot interpret the read-only hint will not throw an exception when asked for a read-only transaction but rather silently ignore the hint.

The propagation attribute can be used to define behavior of the target methods: if they should be executed in an existing or new transaction, or no transaction at all. Default: PROPAGATION\_REQUIRED.

The isolation attribute value defines how data modified in a transaction affects other simultaneous transactions. As a general idea, transactions should be isolated. A transaction should not be able to access changes from another uncommitted transaction. ISOLATION\_DEFAULT: the default isolation level of the DBMS. Exclusively designed for use with Propagation.REQUIRED or Propagation.REQUIRES\_NEW since it only applies to newly started transactions.

***timeout***. The timeout for this transaction (in seconds). Defaults to the default timeout of the underlying transaction system (transaction manager provider). Exclusively designed for use with Propagation.REQUIRED or Propagation.REQUIRES\_NEW since it only applies to newly started transactions.

***rollbackFor.*** Defines zero (0) or more exception classes, which must be subclasses of Throwable, indicating which exception types must cause a transaction rollback. By default, it’s rolled back only when a RuntimeException or Error is thrown. In using this attribute, the rollback can be triggered for checked exceptions as well.

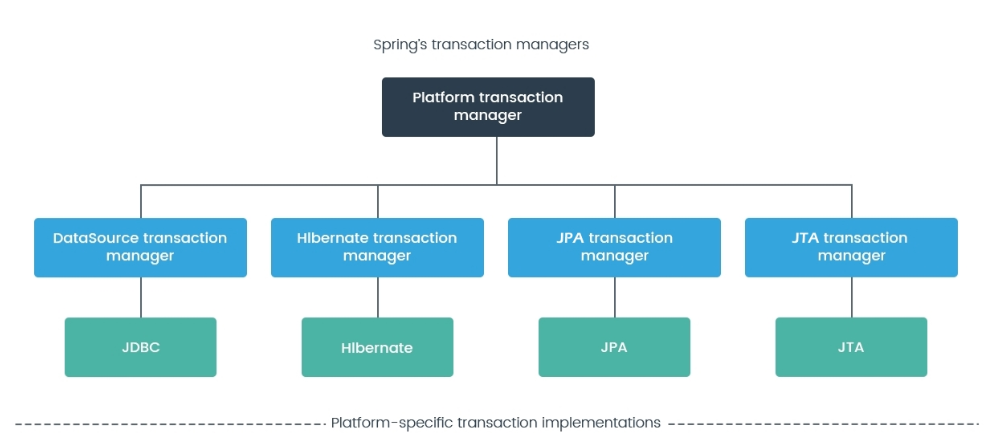
***noRollbackFor*** attribute values should be one or more exception classes, subclasses of Throwable. When this type of exception is thrown during the execution of a transactional method, the transaction is not rolled back. By default, a transaction is rolled back only when a RuntimeException is thrown. Using this attribute, rollback of a transaction can be avoided for a RuntimeException as well.

**What is the PlatformTransactionManager?**

PlatformTransactionManager is the central interface in Spring's transaction infrastructure. Applications can use this directly, but it is not primarily meant as API: Typically, applications will work with either TransactionTemplate or declarative transaction demarcation through AOP.

For implementors, it is recommended to derive from the provided AbstractPlatformTransactionManager class, which pre-implements the defined propagation behavior and takes care of transaction synchronization handling. Subclasses have to implement template methods for specific states of the underlying transaction, for example: begin, suspend, resume, commit.

The default implementations of this strategy interface are JtaTransactionManager and DataSourceTransactionManager, which can serve as an implementation guide for other transaction strategies.



It uses the TransactionDefinition and TransactionStatus interfaces to create and manage transactions. A transaction manager is needed no matter which transaction management strategy (programmatic or declarative) you choose.

***Public interface PlatformTransactionManager(){***

***// The TransactionDefinition interface controls the properties of a transaction -- Propagation, Isolation, Timeout, Read-only***

***// The returned TransactionStatus is used to control the transaction execution.***

***TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;***

***//commit(): Commit the given transaction, with regard to its status***

***Void commit(TransactionStatus status) throws TransactionException;***

***// Perform a rollback of the given transaction***

***Void rollback(TransactionStatus status) throws TransactionException;***

***}***

***PlatformTransactionManager implementations:***

You typically define PlatformTransactionManager implementation through dependency injection.

Deal with only a single data source in your application and access it with JDBC, use DataSourceTransactionManager.

If you are using an object-relational mapping framework to access a database, you should choose a corresponding transaction manager for this framework, such as HibernateTransactionManager or JpaTransactionManager.

If you are using JTA for transaction management on a Java EE application server, you should use JtaTransactionManager to look up a transaction from the application server. Additionally, JtaTransactionManager is appropriate for distributed transactions (transactions that span multiple resources).

**What is a transaction isolation level? How many do we have and how are they ordered?**

***Isolation:*** Isolation defines how protected your uncommitted data is to other concurrent transactions. Isolation levels range from least protective, which offers access to uncommitted data, to most protective, at which no two transactions work at the same time. Isolation is closely related to concurrency and consistency. If you increase the level of isolation, you get more consistency but lose concurrency—that is, performance. On the other hand, if you decrease the level, your transaction performance increases, but you risk losing consistency. In a nutshell, transaction isolation is the degree to which particular transaction is isolated from other transactions. There are 4 (read\_uncomitted, read\_committed, repeatable\_read, serializable) ordered from lowest to highest.

Isolation and concurrency are inversely proportional to each other.

***1) DEFAULT*** - Use the default isolation level of the underlying datastore. All other levels correspond to the JDBC isolation levels.

***2) READ\_UNCOMMITTED*** - A constant indicating that dirty reads, non-repeatable reads and phantom reads can occur. This level allows a row changed by one transaction to be read by another transaction before any changes in that row have been committed (a "dirty read"). If any of the changes are rolled back, the second transaction will have retrieved an invalid row.

***3) READ\_COMMITTED*** - A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur. This level only prohibits a transaction from reading a row with uncommitted changes in it.

***4) REPEATABLE\_READ*** - A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur. This level prohibits a transaction from reading a row with uncommitted changes in it, and it also prohibits the situation where one transaction reads a row, a second transaction alters the row, and the first transaction rereads the row, getting different values the second time (a "non-repeatable read").

***5) SERIALIZABLE*** - A constant indicating that dirty reads, non-repeatable reads and phantom reads are prevented. This level includes the prohibitions in REPEATABLE\_READ and further prohibits the situation where one transaction reads all rows that satisfy a WHERE condition, a second transaction inserts/deletes a row that satisfies that WHERE condition, and the first transaction rereads for the same condition, retrieving the additional "phantom" row in the second read.

SERIALIZABLE isolation level is the most restrictive of all isolation levels. Transactions are executed with locking at all levels (read, range and write locking) so they appear as if they were executed in a serialized way. This leads to a scenario where none of the issues mentioned above (dirty reads, non-repeatable reads and phantom reads) may occur, but in the other way we don't allow transaction concurrency and consequently introduce a performance penalty.

It's important to mention that setting a the right isolation level in the datasource or datastore's defaults instead of forcing to explicit isolation level in annotations, together with the right connection string parameters in MySQL (so that it doesn't change always isolation level) can boost performance quite a lot.

**What does transaction propagation mean?**

Transaction propagation defines whether current transaction will be extended or not.

It is to define behavior of the target methods: if they should be executed in an existing or new transaction, or no transaction at all.

Transaction propagation behaviors for use with the Transactional annotation can be easily understood with nested method calls scenario.

***1) REQUIRED*** - Support a current transaction, create a new one if none exists. This is the default setting of a transaction annotation. If the second method throws an exception that causes rollback, the whole transaction rolls back. It doesn’t matter if the first transaction handles that exception or not.

***2) REQUIRES\_NEW*** - Create a new transaction, and suspend the current transaction if one exists. If the second method throws an exception that causes rollback of operations ONLY in that method, and the whole transaction do not roll back.

***3) MANDATORY*** - Support a current transaction, throw an exception if none exists.

***4) NESTED*** - Execute within a nested transaction if a current transaction exists, behave like REQUIRED otherwise. A nested transaction is a database transaction that is started by an instruction within the scope of an already started transaction.

***5) NOT\_SUPPORTED*** - Execute non-transactionally, suspend the current transaction if one exists.

***6) NEVER*** - Execute non-transactionally, throw an exception if a transaction exists.

**Is the JDBC template able to participate in an existing transaction?**

If you define a method as @Transactional and internally add some JdbcTemplate code it will run in that transaction; but JdbcTemplate itself cannot manage transactions - that is job of TransactionTemplate.

JDBC template is able to participate in an existing transaction both declarative and programmatic ways, by wrapping the DataSource using a TransactionAwareDataSourceProxy. This is a proxy for a target DataSource, which wraps the target DataSource to add awareness of Spring-managed transactions.

**What is @EnableTransactionManagement for?**

EnableTransactionManagement enables Spring's annotation-driven transaction management capability. To be used on @Configuration classes as follows:

***@Configuration***

***@EnableTransactionManagement***

***public class AppConfig {***

***@Bean***

***public FooRepository fooRepository() {***

***// configure and return a class having @Transactional methods***

***return new JdbcFooRepository(dataSource());***

***}***

***@Bean***

***public DataSource dataSource() {***

***// configure and return the necessary JDBC DataSource***

***}***

***@Bean***

***public PlatformTransactionManager txManager() {***

***return new DataSourceTransactionManager(dataSource());***

***}***

***}***

For reference, the example above can be compared to the following Spring XML configuration:

<beans>

<tx:annotation-driven/>

<bean id="fooRepository" class="com.foo.JdbcFooRepository">

<constructor-arg ref="dataSource"/>

</bean>

<bean id="dataSource" class="com.vendor.VendorDataSource"/>

<bean id="transactionManager" class="org.sfwk...DataSourceTransactionManager">

<constructor-arg ref="dataSource"/>

</bean>

</beans>

In both of the scenarios above, @EnableTransactionManagement and <tx:annotation-driven/> are responsible for registering the necessary Spring components that power annotation-driven transaction management.

Components registered when the @EnableTransactionManagement annotation is used are:

1) A TransactionInterceptor: Calls @Transactional methods

2) The Proxy or AspectJ based advice, intercepts methods annotated with @Transactional

**What happens if one @Transactional annotated method is calling another @Transactional annotated method on the same object instance?**

As per the limitation of Spring AOP, a self-invocation of a proxied Spring Bean effectively bypasses the proxy, thus the second method will be executed in the same transaction with the first.

**Service layer or Repository layer?**

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.

**Where can the @Transactional annotation be used? What is a typical usage if you put it at class level?**

@Transactional annotation be used on an individual method, an interface or on a class.

At the class level, this annotation applies as a default to all methods of the declaring class and its subclasses. Note that it does not apply to ancestor classes up the class hierarchy; methods need to be locally redeclared in order to participate in a subclass-level annotation.

At the method level, it can be applied only to Public methods. For protected, private or package-visible methods with the @Transactional annotation, no error is raised, but the annotated method does not exhibit the configured transactional settings.

If you need to annotate non-public methods, consider using AspectJ.

**What does declarative transaction management mean?**

***Declarative transaction management:***

1) It keeps transaction management out of business logic.

2) Easy to configure. Use Java annotations or XML configuration files.

3) Basically, when those specified methods are called, Spring begins a new transaction, and when the method returns without any exception it commits the transaction; otherwise, it rolls back. Hence, you don’t have to write a single line of transaction demarcation code in your method bodies.

***How It Works -***

1) The @EnableTransactionManagement annotation activates annotation-based declarative transaction management.

2) Spring Container scans managed beans' classes for the @Transactional annotation.

3) When the annotation is found, it creates a proxy that wraps your actual bean instance. From now on, that proxy instance becomes your bean, and it’s delivered from Spring Container when requested.

***Programmatic transaction management is a good idea only if:***

1) Has only a small number of transactional operations. For example, if you have a web application that requires transactions only for certain update operations, you may not want to set up transactional proxies by using Spring or any other technology.

2) Being able to set the transaction name explicitly.

**What is the default rollback policy? How can you override it?**

Default rollback policy: only when a RuntimeException or Error is thrown.

Default settings can be overridden with rollbackFor and/or noRollbackFor attributes of @Transactional annotation.

***rollbackFor***. Defines zero (0) or more exception classes, which must be subclasses of Throwable, indicating which exception types must cause a transaction rollback. By default, it’s rolled back only when a RuntimeException or Error is thrown. In using this attribute, the rollback can be triggered for checked exceptions as well.

***noRollbackFor*** attribute values should be one or more exception classes, subclasses of Throwable. When this type of exception is thrown during the execution of a transactional method, the transaction is not rolled back. By default, a transaction is rolled back only when a RuntimeException is thrown. Using this attribute, rollback of a transaction can be avoided for a RuntimeException as well.

**What is the default rollback policy in a JUnit test, when you use the @RunWith(SpringJUnit4ClassRunner.class) in JUnit 4 or @ExtendWith(SpringExtension.class) in JUnit 5, and annotate your @Test annotated method with @Transactional?**

1) Test-methods will be executed in a transaction, and will roll back after completion.

2) The rollback policy of a test can be changed using the @Rollback set to false, @Rollback(false)

3) @Commit indicates that the transaction for a transactional test method should be committed after the test method has completed. You can use @Commit as a direct replacement for @Rollback(false) to more explicitly convey the intent of the code. Analogous to @Rollback, @Commit can also be declared as a class-level or method-level annotation.

4) The @DataJpaTest tests are transactional and rolled back at the end of each test by default. You can disable this default rollback behavior for a single test or for an entire test class by annotating with @Transactional(propagation = Propagation.NOT\_SUPPORTED).

**Why is the term "unit of work" so important and why does JDBC AutoCommit violate this pattern?**

The unit of work describes the atomicity of transactions. JDBC AutoCommit will cause each individual SQL statement as to be executed in its own transaction, which makes it impossible to perform operations that consist of multiple SQL statements as a unit of work. JDBC AutoCommit can be disabled by calling the setAutoCommit() to false on a JDBC connection.

**Spring JPA**

**What do you need to do in Spring if you would like to work with JPA?**

***1) Declare dependencies***: ORM dependency, db driver dependency, transaction manager dependency. To use Spring Data components in a JPA project, a dependency on the package spring-data-jpa must be introduced.

***2) @Entity classes -***

* Is part of the javax.persistence.\*, not JPA!
* @Entity marks classes as templates for domain objects, also called entities to database tables.
* The @Entity annotation can be applied only at class level.
* @Entity are mapped to database tables matching the class name, unless specified otherwise using the @Table annotation.
* @Entity and @Id are mandatory for a domain class.

3) Define an EntityManagerFactory bean.

You can define either an application managed entity manager(LocalEntityManagerFactoryBean) or container managed entity manager (LocalContainerEntityManagerFactoryBean).

Simplest: LocalEntityManagerFactoryBean. It produces an application-managed EntityManagerFactory.

Obtain an EntityManagerFactory using JNDI, use when app ran in Java EE server

***Full JPA capabilities:*** LocalContainerEntityManagerFactoryBean. You must create LocalContainerEntityManagerFactoryBean and not EntityManagerFactory directly, since the former also participates in exception translation mechanisms in addition to creating EntityManagerFactory .

4) Define a DataSource bean

5) Define a TransactionManager bean - JpaTransactionManager (PlatformTransactionManager)

6) Set EntityManagerFactory in JpaTransactionManager and set dataSource in EntityManagerFactory.

7) Implement repositories. A custom repository interface extends JpaRepository.

**Are you able to participate in a given transaction in Spring while working with JPA?**

Yes you can.

The Spring JpaTransactionManager supports direct DataSource access within one and the same transaction allowing for mixing plain JDBC code that is unaware of JPA with code that use JPA. Spring JPA also allows a configured JpaTransactionManager to expose a JPA transaction to JDBC access code that accesses the same DataSource.

**Which PlatformTransactionManager(s) can you use with JPA?**

JpaTransactionManager or JtaTransactionManager

Implementations of PlatformTransactionManager interface. E.g.,

1) DataSourceTransactionManager: Suitable if you are only using JDBC

2) HibernateTransactionManager

* Hibernate without JPA
* Also possible to use JDBC at the same time

3) JpaTransactionManager:

* Suitable if you are using JPA.
* Also possible to use JDBC at the same time

4) JtaTransactionManager

Suitable if you are using global transactions—that is, the distributed transaction management capability of your application server.

You can use any data access technology

When using JPA with one single entity manager factory, the Spring Framework JpaTransactionManager is the recommended choice. This is also the only transaction manager that is JPA entity manager factory aware.

JtaTransactionManager is used for global transactions, so that they can span multiple resources such as databases, queues etc. If the application has multiple JPA entity manager factories that are to be transactional, then a JTA transaction manager is required.

5) JdoTransactionManage

* using JDO
* Also possible to use JDBC at the same time

6) WebLogicJtaTransactionManager

**What do you have to configure to use JPA with Spring? How does Spring Boot make this easier?**

For Spring - Refer to question -- What do you need to do in Spring if you would like to work with JPA?

***JPA in SpringBoot-***

1) SpringBoot provides a default set of dependencies needed for JPA in starter.

2) Provides all default Spring beans needed to use JPA.

3) Provides a number of default properties related to persistence and JPA.

4) Sometimes it is useful to disable Spring Data Auto Configuration in SpringBoot.

***Disable Using Annotation***

***@SpringBootApplication(exclude = {***

***DataSourceAutoConfiguration.class,***

***DataSourceTransactionManagerAutoConfiguration.class,***

***HibernateJpaAutoConfiguration.class} )***

***Disable Using Property File***

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,

org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,

org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration

**What does @PersistenceContext do?**

***PersistenceContext in general -***

1) Entities are managed by javax.persistence.EntityManager instance using persistence context.

2) Each EntityManager instance is associated with a persistence context.

3) Within the persistence context, the entity instances and their lifecycle are managed.

4) Persistence context defines a scope under which particular entity instances are created, persisted, and removed.

5) A persistence context is like a cache which contains a set of persistent entities, So once the transaction is finished, all persistent objects are detached from the EntityManager's persistence context and are no longer managed.

6) Default persistence context duration is one single transaction. However, it can be configured.

7) The persistence context itself is managed by EntityManager.

8) It’s essentially a Cache, containing a set of domain objects/entities in which for every persistent entity there is a unique entity instance.

***@PersistenceContext annotation -***

1) Used for entity manager injection.

2) Expresses a dependency on a container-managed EntityManager and its associated persistence context.

3) This field does not need to be autowired, since this annotation is picked up by an infrastructure Spring bean postprocessor bean that makes sure to create and inject an EntityManager instance.

4)@PersistenceContext has a type attribute

a) ***PersistenceContextType.TRANSACTION*** - In stateless beans, like singleton bean, it is safe to use only the PersistenceContextType.TRANSACTION value for a shared EntityManager to be created and injected into for the current active transaction.

b) ***PersistenceContextType.EXTENDED*** is purposefully designed to support beans, like stateful EJBs, session Spring beans, or request-scoped Spring beans. The shared EntityManager instance wouldn’t be bound to the active transaction and might span more than one transaction.

**How a custom repository implementation can add support for Jdbc, JPA and Hibernate?**

1) JDBC Support typical JDBC support. You could have the DataSource injected into an initialization method, where you would create a JdbcTemplate and other data access support classes

***@Repository***

***public class JdbcMovieFinder implements MovieFinder {***

***private JdbcTemplate jdbcTemplate;***

***@Autowired***

***public void init(DataSource dataSource) {***

***this.jdbcTemplate = new JdbcTemplate(dataSource);***

***}***

***}***

2) JPA repository JPA-based repository needs access to an EntityManager.

***@Repository***

***public class JpaMovieFinder implements MovieFinder {***

***@PersistenceContext***

***private EntityManager entityManager;***

***}***

3) classic Hibernate APIs inject SessionFactory

***@Repository***

***public class HibernateMovieFinder implements MovieFinder {***

***private SessionFactory sessionFactory;***

***@Autowired***

***public void setSessionFactory(SessionFactory sessionFactory) {***

***this.sessionFactory = sessionFactory;***

***}***

***}***

**Spring Data JPA**

**What is a Repository interface?**

***Repository Interface -*** Central repository marker interface, that is, it does not define any methods. Domain repositories extending this interface can selectively expose CRUD methods by simply declaring methods of the same signature as those declared in CrudRepository.

Spring data uses the Repository interface to discover repositories defined in the application.

***@Indexed***

***public interface Repository<T,ID>***

***Type Parameters:***

* T - the domain type the repository manages
* ID - the type of the id (primary key) of the entity the repository manages

Common Known Subinterfaces: (Spring-Specialized Interfaces)

CrudRepository<T,ID>, PagingAndSortingRepository<T,ID>

@NoRepositoryBean

public interface CrudRepository<T,ID> extends Repository<T,ID>

Interface for generic CRUD operations on a repository for a specific type.

@NoRepositoryBean

public interface PagingAndSortingRepository<T,ID> extends CrudRepository<T,ID>

Extension of CrudRepository to provide additional methods to retrieve entities using the pagination and sorting abstraction.

A Spring Data repository is also known as a “instant” repository, because they can be created instantly by extending one of the Spring-specialized interfaces like JpaRepository, PagingAndSortingRepository or CrudRepository.

When a custom repository interface extends CrudRepository or JpaRepository, it will automatically be enriched with functionality to save entities, search them by ID, retrieve all of them from the database, delete entities, flush, etc.

By default, repositories are instantiated eagerly unless explicitly annotated with @Lazy. LAZY is a decent choice for testing scenarios.

**How do you define a Repository interface?**

Typically, your repository interface will extend Repository, CrudRepository or PagingAndSortingRepository. Alternatively, if you do not want to extend Spring Data interfaces, you can also annotate your repository interface with @RepositoryDefinition.

A Spring Data repository interface is also known as a “instant” repository, because they can be created instantly using any of the above 2 ways.

***public interface FixedDepositRespository extends Repository<FixedDepositDetails,Integer> {***

***FixedDepositDetails save (FixedDepositDetails entity);***

***FixedDepositDetails findById(Integer id);***

***}***

Or

***@RepositoryDefinition(domainClass=FixedDepositDetails.class, idClass=Integer.class)***

***public interface FixedDepositRespository {***

***FixedDepositDetails save (FixedDepositDetails entity);***

***FixedDepositDetails findById(Integer id);***

***}***

When a custom repository interface extends CrudRepository, PagingAndSortingRepository or JpaRepository, it will automatically be enriched with functionality to save entities, search them by ID, retrieve all of them from the database, delete entities, flush, etc.

Use of @Repository annotation is optional. You don't need @Repository at all in combination with Spring Data interfaces. @Repository is a meta annotation which you add to implementations of a repository layer, not interfaces.

@Repository annotation indicates that an annotated class is a "Repository". As of Spring 2.5, this annotation also serves as a specialization of @Component, allowing for implementation classes to be autodetected through classpath scanning.

// @Repository annotation NOT REQUIRED HERE. It should be on implementing class for interface AuditEmployeeRepository (if any)

public interface AuditEmployeeRepository extends PagingAndSortingRepository<AuditEmployee, AuditEmployeeId>

**Why is Repository interface an interface and not a class?**

Spring data repository provides standard implementations for commonly used functionalities in a repository interfaces like JpaRepository, PagingAndSortingRepository or CrudRepository. This way we don't have to worry about implementation part. Our custom interface just needs to selectively extend these interfaces depending on required functions. We do not need a class here because with class implementing these interfaces, we must implement methods, which is not required here since mostly we use Repository interface as a “instant” repository where common functionality is already implemented.

If we want to add custom methods into all repositories, we have to replace the default repository implementation (SimpleJpaRepository) with our own repository implementation. Otherwise, most commonly we make use of common functionalities provided by default.

Hence interface is a good choice for Repository interface.

**What is the naming convention for finder methods in a Repository interface?**

Repository methods are named following a pattern that helps Spring data generate queries.

You can declare query methods in your custom repository interface to query the data store. Query method names have format like findxxxBy, readxxxBy, getxxxBy, queryxxxBy, countxxxBy etc. Spring Data comes with sophisticated Query Builder that builds queries specific to the data store based on the query methods declared in the repository.

For e.g. - Method - readSpitterByFirstnameOrLastnameOrderByLastName() can be categorized as below read<Query Verb> Spitter<Subject> By FirstnameOrLastnameOrderByLastName<Predicate>()

There are 3 main components - Query Verb, Subject, Predicate and Limiting Query Results.

***1) Query Verb -***

verbs in the method name: get, read, find, query, stream and count.

a) findAllByXxx() and findByXxx() identical. - They return all rows with specified criteria

b) The count verb, on the other hand, returns a count of matching objects, rather than the objects themselves.

***2) Subject -***

a) The subject of a repository method is optional.

b) Below methods are all equal! Because the type of object being retrieved is determined by how you parameterize the JpaRepository interface, not the subject of the method name.

readSpittersByFirstnameOrLastname() = readByFirstnameOrLastname()

readPuppiesByFirstnameOrLastname() = readThoseThingsWeWantByFirstnameOrLastname()

c) There is one exception to the subject being ignored. If the subject starts with the word Distinct, then the generated query will be written to ensure a distinct result set.

***3) Predicate -***

a) The predicate specifies the properties that will constrain the result set.

b) Each condition must reference a property and may also specify a comparison operation.

c) If the comparison operator is left off, it’s implied to be an equals operation.

d) You may choose any other comparison operations, When dealing with String properties, the condition may also include IgnoringCase or IgnoresCase.

e) You may also use AllIgnoringCase or AllIgnoresCase after all the conditions to ignore case for all conditions.

f) Conditional parts are separated by either And or Or.

***4) Limiting Query Results -***

a) The results of query methods can be limited by using the first or top keywords.

b) An optional numeric value can be appended to top or first to specify the maximum result size to be returned.

c) If the number is left out, a result size of 1 is assumed. java User findFirstByOrderByLastnameAsc(); User findTopByOrderByAgeDesc();

***Examples --***

* Page queryFirst10ByLastname(String lastname, Pageable pageable);
* Slice findTop3ByLastname(String lastname, Pageable pageable);
* List findFirst10ByLastname(String lastname, Sort sort);
* List findTop10ByLastname(String lastname, Pageable pageable);

**What is @NoRepositoryBean annotation?**

Annotation to exclude repository interfaces from being picked up and thus in consequence getting an instance being created.

This will typically be used when providing an extended base interface for all repositories in combination with a custom repository base class to implement methods declared in that intermediate interface. In this case you typically derive your concrete repository interfaces from the intermediate one but don't want to create a Spring bean for the intermediate interface.

***1) Defining a Base Repository Interface***

***@NoRepositoryBean***

***public interface ExtendedRepository<T, ID extends Serializable> extends JpaRepository<T, ID> {***

***public List<T> findByAttributeContainsText(String attributeName, String text);***

***}***

Our interface extends the JpaRepository interface so that we'll benefit from all the standard behavior.

You'll also notice we added the @NoRepositoryBean annotation. This is necessary because otherwise, the default Spring behavior is to create an implementation for all subinterfaces of Repository.

Here, we'll want to provide our implementation that should be used, as this is only an interface meant to be extended by the actual entity-specific DAO interfaces.

***2) Implementing a Base Class***

Next, we'll provide our implementation of the ExtendedRepository interface:

***public class ExtendedRepositoryImpl<T, ID extends Serializable>***

***extends SimpleJpaRepository<T, ID> implements ExtendedRepository<T, ID> {***

***private EntityManager entityManager;***

***public ExtendedRepositoryImpl(JpaEntityInformation<T, ?>***

***entityInformation, EntityManager entityManager) {***

***super(entityInformation, entityManager);***

***this.entityManager = entityManager;***

***}***

***@Transactional***

***public List<T> findByAttributeContainsText(String attributeName, String text) {***

***CriteriaBuilder builder = entityManager.getCriteriaBuilder();***

***CriteriaQuery<T> cQuery = builder.createQuery(getDomainClass());***

***Root<T> root = cQuery.from(getDomainClass());***

***cQuery***

***.select(root)***

***.where(builder***

***.like(root.<String>get(attributeName), "%" + text + "%"));***

***TypedQuery<T> query = entityManager.createQuery(cQuery);***

***return query.getResultList();***

***}***

***}***

This class extends the SimpleJpaRepository class, which is the default class that Spring uses to provide implementations for repository interfaces.

This requires that we create a constructor with the JpaEntityInformation and EntityManager parameters that calls the constructor from the parent class. We also need the EntityManager property to use in our custom method.

Also, we have to implement the custom method inherited from the ExtendedRepository interface:

***3) JPA Configuration***

To tell Spring to use our custom class instead of the default one for building repository implementations, we can use the repositoryBaseClass attribute:

***@Configuration***

***@EnableJpaRepositories(basePackages = "org.baeldung.persistence.dao",***

***repositoryBaseClass = ExtendedRepositoryImpl.class)***

***public class StudentJPAH2Config {***

***// additional JPA Configuration***

***}***

4) Creating an Entity Repository

First, let's add a simple Student entity:

***@Entity***

***public class Student {***

***@Id***

***private long id;***

***private String name;***

***// standard constructor, getters, setters***

***}***

Then, we can create a DAO for the Student entity which extends the ExtendedRepository interface:

***public interface ExtendedStudentRepository extends ExtendedRepository<Student, Long> {***

***}***

And that's it! Now our implementation will have the custom findByAttributeContainsText() method. Similarly, any interface we define by extending the ExtendedRepository interface will have the same method.

**How can you restrict default methods to callers exposed by a repository?**

1) Create the repository by extending CrudRepository and declare only those methods in your custom repository interface that you want to make available to the callers.

2) Create the repository by extending Repository interface and declare only those CrudRepository methods in your custom repository interface that you want to make available to the callers.

***public interface FixedDepositRespository extends Repository<FixedDepositDetails,Integer> {***

***FixedDepositDetails save (FixedDepositDetails entity);***

***FixedDepositDetails findById(Integer id);***

***}***

In above code, only save and findById methods are exposed to caller.

Notice that signatures of save and findById methods match signatures of these methods declared in CrudRepository interface.

We don't need to implement these methods as implementation is provide by Spring Data.

**What is JpaRepository?**

Repository, CrudRepository or PagingAndSortingRepository interfaces provided by Spring Data are agnostic to the underlying data store, that is you can't use them to take advantage of features specific to the data store. For this reason, Spring Data project defines a data store specific repository interface like JpaRepository for Spring Data JPA or MongoRepository for Spring Data MongoDB.

JpaRepository - JPA specific extension of Repository.

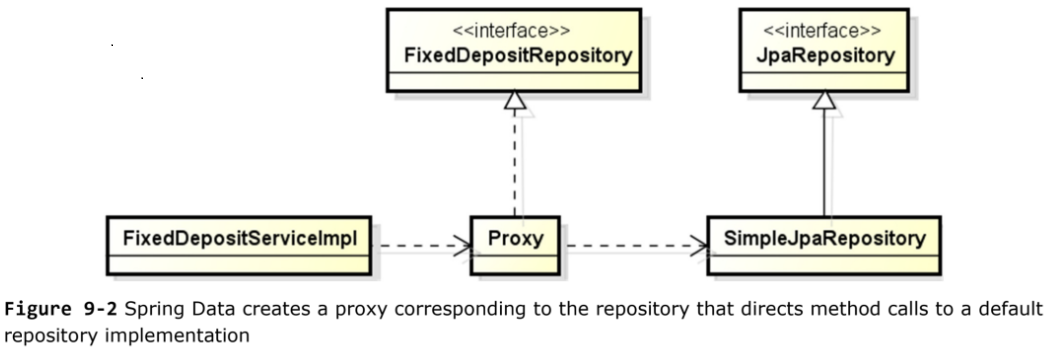
@NoRepositoryBean

***public interface JpaRepository<T,ID> extends PagingAndSortingRepository<T,ID>, QueryByExampleExecutor<T>***

**How are Spring Data repositories implemented by Spring at runtime?**

For a Spring Data repository a JDK dynamic proxy is created which intercepts all calls to the repository.

The default behavior is to route calls to the default repository implementation, which in Spring Data JPA is the SimpleJpaRepository class.



Spring Data creates a proxy corresponding to each repository interface that you define in your application. For instance, a proxy is created corresponding to the custom "FixedDepositRespository" interface. The proxy holds reference to the default repository implementation that is provided out-of-the-box by Spring Data. In case of Spring Data JPA, the default repository implementation is an instance of SimpleJpaRepository class (an implementation of JpaRepository interface) that uses JPA for accessing relational data store. Method calls to FixedDepositRespository interface are intercepted by the proxy and delegated to SimpleJpaRepository instance.

For instance, when you call FixedDepositRespository's save method, proxy delegates the call to SimpleJpaRepository's save method.

**What is @Query used for?**

Spring @Query annotation is used to customize methods(queries) by specifying a query to be used with a Spring Data JPA repository method. If a query is quite complex, you can use @Query annotation to explicitly specify the query.

***@Query("select fd from FixedDepositDetails fd where fd.tenure = ?1 and fd.fdAmount <= ?2 and fd.active = ?3")***

***List<FixedDepositDetails> findByCustomQuery(int tenure, int fdAmount, String active);***

In the above example, @Query annotation specifies the JPQL query (a platform independent query language) to be executed. ?1, ?2 and ?3 refer to the arguments passed to the findByCustomQuery method.

@EnableJpaRepositories’s 'queryLookupStrategy' attribute specifies whether Spring Data JPA derives query from the query method name or directly uses the query specified by @Query annotation. By default, Spring Data JPA creates a query from the method name only if no @Query annotation is specified for the method.

When the name of the named parameter is the same as the name of the argument in the method annotated with @Query, the @Param annotation is not needed. But if the method argument has a different name, the @Param annotation is needed to tell Spring that the value of this argument is to be injected in the named parameter in the query.

Queries annotated to the query method take precedence over queries defined using @NamedQuery or named queries declared in orm.xml.

Annotation-based configuration has the advantage of not needing another configuration file to be edited, lowering maintenance effort.

***public interface UserRepo extends JpaRepository<User, Long> {***

***@Query("select u from User u where u.username like %?1%")***

***List<User> findAllByUserName(String username);***

***// using named parameters***

***@Query("select u from User u where u.username= :un")***

***User findOneByUsername(@Param("un") String username);***

***@Query("select u.username from User u where u.id= :id")***

***String findUsernameById(Long id);***

***@Query("select count(u) from User u")***

***long countUsers();***

***}***

Named queries are part of the metadata, and are defined with the annotation @NamedQuery, The annotation @NamedQueries can be used to group multiple queries together.

***@Entity***

***@Table(name="P\_USER")***

***@SequenceGenerator(name = "seqGen", allocationSize = 1) @NamedQueries({***

***@NamedQuery(name=User.FIND\_BY\_USERNAME\_EXACT, query = "from User u where username= ?"), @NamedQuery(name=User.FIND\_BY\_USERNAME\_LIKE, query = "from User u where username like ?")***

***})***

***public class User extends AbstractEntity { }***

**What are JPQL, native, named and Named-Native queries in spring data JPA?**

The @Query annotation in Spring Data JPA to execute both JPQL and native SQL queries. we can annotate the method with the @Query annotation — its value attribute contains the JPQL or SQL to execute. The @Query annotation takes precedence over named queries, which are annotated with @NamedQuery or defined in an orm.xml file.

***1. JPQL***

By default the query definition uses JPQL.

***@Query("SELECT u FROM User u WHERE u.status = 1")***

***Collection<User> findAllActiveUsers();***

When we use JPQL for a query definition, then Spring Data can handle pagination/sorting without any problem.

***2. Native***

We can use also native SQL to define our query. All we have to do is to set the value of the nativeQuery attribute to true and define the native SQL query in the value attribute of the annotation:

***@Query(***

***value = "SELECT \* FROM USERS u WHERE u.status = 1",***

***nativeQuery = true)***

***Collection<User> findAllActiveUsersNative();***

Cannot use native queries with dynamic sorting and/or pagination. However, We can enable pagination for native queries by declaring an additional attribute countQuery — this defines the SQL to execute to count the number of rows in the whole result:

***@Query(***

***value = "SELECT \* FROM Users ORDER BY id",***

***countQuery = "SELECT count(\*) FROM Users",***

***nativeQuery = true)***

***Page<User> findAllUsersWithPagination(Pageable pageable);***

***Native query vs named query***

1. Native query refers to actual sql queries (referring to actual database objects). These queries are the sql statements which can be directly executed in database using a database client.

2. Named query is the way you define your query by giving it a name. You could define this in mapping file in hibernate or also using annotations at entity level.

***@NamedNativeQuery -***

Named SQL queries are defined using the @NamedNativeQuery annotation. This annotation may be placed on any entity and defines the name of the query as well as the query text.

***@Entity(name="EmployeeEntity")***

***@Table (name="employee")***

***@NamedNativeQueries({***

***@NamedNativeQuery(***

***name = "getAllEmployees",***

***query = "SELECT id, firstName, lastName, email, department.id, department.name " +***

***"FROM employee, department",***

***resultClass=EmployeeEntity.class***

***),***

***@NamedNativeQuery(***

***name = "getAllEmployeesByDeptId",***

***query = "SELECT id, firstName, lastName, email, department.id, department.name " +***

***"FROM employee, department " +***

***"WHERE department.id = ?",***

***resultClass=EmployeeEntity.class***

***)***

***})***

To execute above SQL queries, you will need to write below code in your DAOImpl class.

***@Override***

***public List<EmployeeEntity> getAllEmployees() {***

***List<EmployeeEntity> employees = manager.createNamedQuery("getAllEmployees", EmployeeEntity.class)***

***.getResultList();***

***return employees;***

***}***

***@Override***

***public List<EmployeeEntity> getAllEmployeesByDeptId(Integer id) {***

***List<EmployeeEntity> employees = manager.createNamedQuery("getAllEmployeesByDeptId", EmployeeEntity.class)***

***.setParameter(1, id)***

***.getResultList();***

***return employees;***

***}***

@NamedNativeQuery is used to write native SQL named queries as above.

@NamedQuery is used to write JPQL named queries as below.

***@Entity***

***@Table(name = "atp\_players")***

***@NamedQueries({***

***@NamedQuery(***

***name = "Player.findAllUnordered",***

***query = "SELECT p FROM Player p"),***

***@NamedQuery(***

***name = "Player.findAllOrderedByName",***

***query = "SELECT p FROM Player p ORDER BY p.name")***

***})***