Spring Transaction Management

In **Spring, transaction management** is a powerful feature that allows you to manage transactional behavior declaratively using annotations like @Transactional. The transaction attributes define the way the transaction should behave under various circumstances.

Here’s a breakdown of the key transaction attributes in **Spring:**

1. Propagation Behavior

The **propagation** of a transaction defines how the transaction should behave when a transactional method is called within the context of an existing transaction. These behaviors are defined using the propagation attribute of @Transactional.

Propagation types:

- **REQUIRED (default):** If a transaction already exists, use it. Otherwise, create a new one.
- **REQUIRES_NEW:** Always create a new transaction, suspending the current one (if any).
- **NESTED:** Execute the method within a nested transaction if a current transaction exists. If no transaction exists, behave like REQUIRED.
- **MANDATORY:** It requires an existing transaction. If no transaction exists, an exception will be thrown.
- **NEVER:** Do not allow a transaction to exist. If a transaction exists, throw an exception.
- **NOT_SUPPORTED:** Do not use a transaction. If a transaction exists, suspend it.
- **SUPPORTS:** If a transaction exists, use it. If none exists, execute the method without a transaction.

2. Isolation Level

The **isolation level** determines how a transaction is isolated from other concurrent transactions, specifically in terms of visibility of data changes. These are the options you can set using the isolation attribute of @Transactional.

Isolation levels:

- **DEFAULT:** Inherits the default isolation level of the underlying database (typically READ_COMMITTED).
- **READ_UNCOMMITTED**: Allows dirty reads (reading uncommitted data from other transactions).
- **READ_COMMITTED:** Ensures no dirty reads, but non-repeatable reads or phantom reads may occur.
- **REPEATABLE_READ:** Prevents dirty reads and non-repeatable reads, but phantom reads are still possible.
- **SERIALIZABLE:** The highest isolation level. Prevents dirty reads, non-repeatable reads, and phantom reads by locking the database, which can severely impact performance.

3. Read-only Status

The **read-only status** is an optimization to indicate that the method will not modify the database. If this flag is set to true, Spring will skip certain optimizations such as the generation of dirty checks.

Usage:

- **readOnly = true:** Marks the transaction as read-only. The underlying database can optimize the operation by not allowing writes.
- **readOnly = false:** Marks the transaction as read-write, allowing modifications.

4. Timeout

The **timeout** specifies the maximum amount of time a transaction can run before being rolled back. If the transaction exceeds the specified timeout, it will be marked as a failure and rolled back.

5. Rollback Policy

The **rollback policy** determines which exceptions will trigger a rollback of the transaction. By default, Spring only rolls back on **unchecked exceptions** (subclasses of **RuntimeException)** and **errors**. However, you can customize this **behavior to rollback** on specific exceptions.

Rollback policies:

- **@Transactional(rollbackFor = Exception.class):** Rollback for specific checked exceptions.
- **@Transactional(noRollbackFor = Exception.class):** Do not **rollback for specific exceptions**.
- The default **behavior is to rollback on RuntimeException and Error.**

Summary of Attributes:

Attribute	**Description**	**Example Value**
propagation	Defines how the transaction behaves if one already exists **(e.g., REQUIRED, REQUIRES_NEW)**	**Propagation.REQUIRED**
isolation	Defines the isolation level for the transaction **(e.g., READ_COMMITTED, SERIALIZABLE)**	**Isolation.READ_COMMITTED**
readOnly	Marks a transaction as read-only, which can optimize performance for read operations	**true**

timeout	Specifies the maximum time a transaction can run before being rolled back	**10 (seconds)**
rollbackFor	Defines the exceptions that will trigger a rollback. By default, it only rolls back on RuntimeException	**CustomException.class**
noRollbackFor	Specifies exceptions that will not trigger a rollback	**IOException.class**

These transaction attributes help to fine-tune transaction management based on the specific needs of your application, ensuring both consistency and performance in your database interactions.

===

The **@Transaction**al annotation in Spring is used to define the transactional behavior for methods or classes that interact with a database. When you apply @Transactional to a method or class, it configures the transaction settings, such as propagation behavior, isolation level, timeout, rollback rules, and read-only status.

Let's break down each of the parameters used in example:

1. propagation = Propagation.REQUIRED

- **Explanation:**
  - The **propagation** setting defines how the transaction should behave in relation to other existing transactions.
  - **Propagation.REQUIRED** means that the method must run within a transaction. If there is an existing transaction, it will join that transaction; if there is no existing transaction, a new one will be started. This is the most commonly used propagation type.

When to use:

  - Use **REQUIRED** when the method should either run within an existing transaction or create a new one if none exists.

- **Other propagation options:**
  - **REQUIRES_NEW:** Always start a new transaction, suspending the current one.
  - **MANDATORY:** Requires an existing transaction. If no transaction exists, an exception is thrown.
  - **SUPPORTS:** If a transaction exists, it will be used; otherwise, no transaction will be used.
  - **NOT_SUPPORTED:** The method should not run within a transaction. If there is an existing transaction, it will be suspended.
  - **NEVER:** The method must not run within a transaction. If a transaction exists, an exception is thrown.
  - **NESTED:** Executes within a nested transaction (if supported by the underlying database).

2. isolation = Isolation.READ_COMMITTED

- **Explanation:**
  - **Isolation** defines the level of visibility each transaction has into the data that is being accessed by other concurrent transactions.
  - **Isolation.READ_COMMITTED** ensures that a transaction can only read data that has been committed by other transactions. This is the default isolation level for most relational databases.
  - **It prevents "dirty reads" (reading data that hasn't been committed)**, but allows "non-repeatable reads" (where a value read by one transaction might change due to another transaction before the first transaction completes).
- **Other isolation levels:**

    - **READ_UNCOMMITTED:** Allows dirty reads, meaning transactions can read uncommitted changes from other transactions.
    - **REPEATABLE_READ:** Prevents dirty reads and non-repeatable reads, but can allow "phantom reads" (new rows inserted by other transactions between reads).
    - **SERIALIZABLE:** Ensures that transactions execute serially, preventing dirty reads, non-repeatable reads, and phantom reads. This is the strictest isolation level and may result in higher locking and decreased concurrency.

3. readOnly = false

- **Explanation:**
    - The **readOnly** flag indicates whether the transaction is meant for read-only operations. Setting it to false means that the transaction will allow both reading and writing to the database.
    - **readOnly = false** tells Spring that the transaction might perform updates, inserts, or deletes in the database. If you only need to read data, setting **readOnly = true** can help optimize the transaction.
- **When to use:**
    - Use **readOnly = true** for methods that only read data and don't modify the database. This can sometimes allow Spring to optimize the transaction.
    - Use **readOnly = false** when the method might modify the database (insert, update, delete).

4. timeout = 10

- **Explanation:**
    - The **timeout** specifies the maximum number of seconds a transaction is allowed to run before being automatically rolled back by the transaction manager. This is useful for preventing long-running transactions from locking resources indefinitely.

    - **timeout = 10** means that if the transaction takes more than 10 seconds, it will be rolled back automatically.

- **When to use:**
    - Use the timeout setting to protect your application from operations that could hang due to database issues, slow queries, or other unexpected delays.
    - A reasonable timeout should be set based on your application's performance needs and the expected query durations.

5. rollbackFor = {CustomException.class}

- **Explanation:**
    - The **rollbackFor** attribute specifies which exceptions should trigger a rollback of the transaction.
    - **rollbackFor = {CustomException.class}** means that if a **CustomException** is thrown within the transactional method, the transaction will be rolled back.
    - By default, Spring only rolls back on unchecked exceptions **(i.e., subclasses of RuntimeException).** If you want to roll back on checked exceptions or custom exceptions, you need to specify them in the rollbackFor attribute.
- **When to use:**
    - Use **rollbackFor** when you want to explicitly define which exceptions should lead to a rollback. This is especially useful when you're handling specific business exceptions or custom error scenarios.

Putting It All Together:

The **@Transactional** annotation in the example configures the transactional behavior as follows:

- **Propagation:** The method must execute within an existing transaction or create a new one if none exists.

- **Isolation Level:** The transaction operates with the **READ_COMMITTED** isolation level, meaning it will not read uncommitted data.
- **Read-Only:** The transaction is not read-only, indicating that data may be modified (inserted, updated, or deleted).
- **Timeout:** The transaction will be automatically rolled back if it exceeds 10 seconds.
- **Rollback on CustomException:** The transaction will be rolled back if a CustomException is thrown.