Optimistic Locking vs Pessimistic Locking-2021-2022

**Optimistic Locking**

[**Optimistic Locking**](http://en.wikipedia.org/wiki/Optimistic_locking)**is a strategy where you read a record, take note of a version number (other methods to do this involve dates, timestamps or checksums/hashes) and check that the version hasn't changed before you write the record back.** When you write the record back you filter the update on the version to make sure it's atomic. (i.e. hasn't been updated between when you check the version and write the record to the disk) and update the version in one hit. If the record is dirty (i.e. different version to yours) you abort the transaction and the user can re-start it. This strategy is most applicable to high-volume systems and three-tier architectures where you do not necessarily maintain a connection to the database for your session. In this situation the client cannot actually maintain database locks as the connections are taken from a pool and you may not be using the same connection from one access to the next.

Optimistic concurrency control transactions involve these phases:[[2]](https://en.wikipedia.org/wiki/Optimistic_concurrency_control#cite_note-KungRobinson1981-2)

* **Begin**: Record a timestamp marking the transaction's beginning.
* **Modify**: Read database values, and tentatively write changes.
* **Validate**: Check whether other transactions have modified data that this transaction has used (read or written). This includes transactions that completed after this transaction's start time, and optionally, transactions that are still active at validation time.
* **Commit/Rollback**: If there is no conflict, make all changes take effect. If there is a conflict, resolve it, typically by aborting the transaction, although other resolution schemes are possible. Care must be taken to avoid a [time-of-check to time-of-use](https://en.wikipedia.org/wiki/Time-of-check_to_time-of-use) bug, particularly if this phase and the previous one are not performed as a single [atomic](https://en.wikipedia.org/wiki/Linearizability) operation.

[**Pessimistic Locking**](http://en.wikipedia.org/wiki/Lock_(database))**is when you lock the record for your exclusive use until you have finished with it.** It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid [Deadlocks](http://en.wikipedia.org/wiki/Deadlock). To use pessimistic locking you need either a direct connection to the database (as would typically be the case in a [two tier client server](http://en.wikipedia.org/wiki/Client-server) application) or an externally available transaction ID that can be used independently of the connection. In the latter case you open the transaction with the TxID and then reconnect using that ID. The DBMS maintains the locks and allows you to pick the session back up through the TxID. This is how distributed transactions using two-phase commit protocols (such as [XA](http://www.opengroup.org/bookstore/catalog/c193.htm) or [COM+ Transactions](http://msdn.microsoft.com/en-us/library/ms687120(VS.85).aspx)) work. **Record locking** is the technique of preventing simultaneous access to data in a [database](https://en.wikipedia.org/wiki/Database), to prevent inconsistent results.

The classic example is demonstrated by two [bank](https://en.wikipedia.org/wiki/Banking) clerks attempting to update the same [bank account](https://en.wikipedia.org/wiki/Bank_account) for two different transactions. Clerks 1 and 2 both retrieve (i.e., copy) the account's [record](https://en.wikipedia.org/wiki/Record_(database)). Clerk 1 applies and saves a transaction. Clerk 2 applies a different transaction to his saved copy, and saves the result, based on the original record and his changes, overwriting the transaction entered by clerk 1. The record no longer reflects the first transaction, as if it had never taken place.

A simple way to prevent this is to [lock the file](https://en.wikipedia.org/wiki/File_locking) whenever a record is being modified by any user, so that no other user can save data. This prevents records from being overwritten incorrectly, but allows only one record to be processed at a time, locking out other users who need to edit records at the same time.

**Optimistic locking doesn't necessarily use a version number. Other strategies include using (a) a timestamp or (b) the entire state of the row itself.** **The Pessimistic Locking strategy is ugly but avoids the need for a dedicated version column**, in cases where you aren't able to modify the schema. Optimistic locking is used when you don't expect many collisions. It costs less to do a normal operation but if the collision DOES occur you would pay a higher price to resolve it as the transaction is aborted.

**Pessimistic locking is used when a collision is anticipated**. The transactions which would violate synchronization are simply blocked.

* **Pessimistic locking is useful if there are a lot of updates and relatively high chances of users trying to update data at the same time**. **For example, if each operation can update a large number of records at a time (the bank might add interest earnings to every account at the end of each month)**, and two applications are running such operations at the same time, they will have conflicts.
* **Pessimistic locking is also more appropriate in applications that contain small tables that are frequently updated.** In the case of these so-called hotspots, conflicts are so probable that optimistic locking wastes effort in rolling back conflicting transactions.
* Optimistic locking is useful if the possibility for conflicts is very low – there are many records but relatively few users, or very few updates and mostly read-type operations.

**Multiple threads update same row in database at a time how to maintain consistence**

There are two possible ways to go.

* Either you choose a pessimistic approach and lock rows, tables or even ranges of rows.
* Or you work with versioned Entities (Optimistic Locking).

**Locking Strategies in Hibernate**

**Optimistic**

Optimistic locking assumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect. Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction rolls back.

**Pessimistic**

Pessimistic locking assumes that concurrent transactions will conflict with each other, and requires resources to be locked after they are read and only unlocked after the application has finished using the data.

**Optimistic**

When your application uses long transactions or conversations that span several database transactions, you can store versioning data, so that if the same entity is updated by two conversations, the last to commit changes is informed of the conflict, and does not override the other conversation's work. This approach guarantees some isolation, but scales well and works particularly well in *Read-Often Write-Sometimes* situations.

Hibernate provides two different mechanisms for storing versioning information, a dedicated version number or a timestamp.

@Entity

**public** **class** Flight **implements** Serializable {

...

    @Version

    @Column(name="OPTLOCK")

**public** Integer getVersion() { ... }

}

Here, the version property is mapped to the OPTLOCK column, and the entity manager uses it to detect conflicting updates, and prevent the loss of updates that would be overwritten by a *last-commit-wins* strategy. To artificially increase the version number, see the documentation for properties LockModeType.OPTIMISTIC\_FORCE\_INCREMENT or LockModeType.PESSIMISTIC\_FORCE\_INCREMENT

If the version number is generated by the database, such as a trigger, use the annotation @org.hibernate.annotations.Generated(GenerationTime.ALWAYS).

**Timestamps are a less reliable way of optimistic locking** than version numbers, but can be used by applications for other purposes as well. Timestamping is automatically used if you the @Version annotation on a Date or Calendar.

@Entity

**public** **class** Flight **implements** Serializable {

...

    @Version

**public** Date getLastUpdate() { ... }

}

**Pessimistic**

**Typically, you only need to specify an isolation level for the JDBC connections and let the database handle locking issues.** If you do need to obtain exclusive pessimistic locks or re-obtain locks at the start of a new transaction, Hibernate gives you the tools you need.

The LockMode class defines the different lock levels that Hibernate can acquire.

|  |  |
| --- | --- |
| LockMode.WRITE | acquired automatically when Hibernate updates or inserts a row. |
| LockMode.UPGRADE | acquired upon explicit user request using SELECT ... FOR UPDATE on databases which support that syntax. |
| LockMode.UPGRADE\_NOWAIT | acquired upon explicit user request using a SELECT ... FOR UPDATE NOWAIT in Oracle. |
| LockMode.READ | acquired automatically when Hibernate reads data under Repeatable Read or Serializable isolation level. It can be re-acquired by explicit user request. |
| LockMode.NONE | The absence of a lock. All objects switch to this lock mode at the end of a Transaction. Objects associated with the session via a call to update() or saveOrUpdate() also start out in this lock mode. |

The explicit user request mentioned above occurs as a consequence of any of the following actions:

* A call to Session.load(), specifying a LockMode.
* A call to Session.lock().
* A call to Query.setLockMode().

**Session.lock()** performs a version number check if the specified lock mode is READ, UPGRADE, or UPGRADE\_NOWAIT. In the case of UPGRADE or UPGRADE\_NOWAIT, SELECT ... FOR UPDATE syntax is used. If the requested lock mode is not supported by the database, Hibernate uses an appropriate alternate mode instead of throwing an exception. This ensures that applications are portable.

**Mapping optimistic locking**

JPA defines support for optimistic locking based on either a version (sequential numeric) or timestamp strategy. To enable this style of optimistic locking simply add the javax.persistence.Version to the persistent attribute that defines the optimistic locking value.

**Versionless optimistic locking**

Although the default @Version property optimistic locking mechanism is sufficient in many situations, sometimes, you need rely on the actual database row column values to prevent **lost updates**. Hibernate supports a form of optimistic locking that does not require a dedicated "version attribute". This is achieved through the use of the [@OptimisticLocking](https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/OptimisticLocking.html) annotation which defines a single attribute of type [org.hibernate.annotations.OptimisticLockType](https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/OptimisticLockType.html).

here are 4 available OptimisticLockTypes:

NONE: optimistic locking is disabled even if there is a @Version annotation present

VERSION (the default): performs optimistic locking based on a @Version as described above

ALL: performs optimistic locking based on *all* fields as part of an expanded WHERE clause restriction for the UPDATE/DELETE SQL statements

DIRTY: performs optimistic locking based on *dirty* fields as part of an expanded WHERE clause restriction for the UPDATE/DELETE SQL statements

## Versionless optimistic locking using OptimisticLockType.ALL

@Entity(name = "Person")

@OptimisticLocking(type = OptimisticLockType.ALL)

@DynamicUpdate

public static class Person {

@Id

private Long id;

@Column(name = "`name`")

private String name;

private String country;

private String city;

@Column(name = "created\_on")

private Timestamp createdOn;

//Getters and setters are omitted for brevity

}

As you can see, all the columns of the associated database row are used in the WHERE clause. If any column has changed after the row was loaded, there won’t be any match, and a StaleStateException or an OptimisticLockException is going to be thrown.

When using OptimisticLockType.ALL, you should also use @DynamicUpdate because the UPDATE statement must take into consideration all the entity property values.

Person person = entityManager.find( Person.class, 1L );

person.setCity( "Washington D.C." );

UPDATE

Person

SET

city=?

WHERE

id=?

AND city=?

AND country=?

AND created\_on=?

AND "name"=?

-- binding parameter [1] as [VARCHAR] - [Washington D.C.]

-- binding parameter [2] as [BIGINT] - [1]

-- binding parameter [3] as [VARCHAR] - [New York]

-- binding parameter [4] as [VARCHAR] - [US]

-- binding parameter [5] as [TIMESTAMP] - [2016-11-16 16:05:12.876]

-- binding parameter [6] as [VARCHAR] - [John Doe]

###### **Versionless optimistic locking using OptimisticLockType.DIRTY**

The OptimisticLockType.DIRTY differs from OptimisticLockType.ALL in that it only takes into consideration the entity properties that have changed since the entity was loaded in the currently running Persistence Context.

@Entity(name = "Person")

@OptimisticLocking(type = OptimisticLockType.DIRTY)

@DynamicUpdate

@SelectBeforeUpdate

public static class Person {

@Id

private Long id;

@Column(name = "`name`")

private String name;

private String country;

private String city;

@Column(name = "created\_on")

private Timestamp createdOn;

//Getters and setters are omitted for brevity

}

Person person = entityManager.find( Person.class, 1L );

person.setCity( "Washington D.C." );

UPDATE

Person

SET

city=?

WHERE

id=?

and city=?

-- binding parameter [1] as [VARCHAR] - [Washington D.C.]

-- binding parameter [2] as [BIGINT] - [1]

-- binding parameter [3] as [VARCHAR] - [New York]

**The main advantage of OptimisticLockType.DIRTY over OptimisticLockType.ALL and the default OptimisticLockType.VERSION used implicitly along with the @Version mapping, is that it allows you to minimize the risk of OptimisticLockException across non-overlapping entity property changes**.

**When using OptimisticLockType.DIRTY, you should also use @DynamicUpdate** because the UPDATE statement must take into consideration all the dirty entity property values, and also the @SelectBeforeUpdate annotation so that detached entities are properly handled by the [Session#update(entity)](https://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/Session.html" \l "update-java.lang.Object-) operation.

## Pessimistic Locking

EntityManagerFactory emf = Persistence.createEntityManagerFactory("MyPersistenceUnit");

EntityManager em = emf.createEntityManager();

Avenger avenger = em.find(Avenger.class,5, LockMode.PESSIMISTIC\_READ);

avenger.setStatus("Alive");

em.save(avenger);

The above code ascertains that an exclusive lock is obtained before the transaction tries to update the record. If any other transaction tries to update the same record it throws ***PessimisticLockException***

**Pitfalls of Pessimistic locking**

The pitfall of pessimistic locking is that it is vulnerable to DB locks. If the transaction, which has acquired the lock fails to release it, it can cause DB locks.

**Optimistic Locking**

EntityManagerFactory emf = Persistence.createEntityManagerFactory("MyPersistenceUnit");

EntityManager em = emf.createEntityManager();

Avenger avenger = em.find(Avenger.class,5, LockMode.OPTIMISTIC);

avenger.setStatus("Alive");

em.save(avenger);

If ‘Thread 1’ and ‘Thread 2’ run at the same time and try to update database *OptmisticLockException*will be thrown. This exception can be caught in the code which can then fetch the same record from DB again and try to update again post making the changes.

**Another way to implement this is by using annotations**

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

**@OptimisticLocking(type=OptimisticLockingType.VERSION\_COLUMN)**

**public class Avenger implements Serializable {**

**private String heroName ;**

**private String status;**

**private String canLiftHammer;**

**@Version**

**private long version;**

**...**

}

Optimistic Locking vs Pessimistic Locking tradeoffs

There are scenarios where pessimistic locking can be quite useful despite its shortcomings. Example - If multiple users are trying to update a shared document, then with optimistic locking, managing updates can become very hard.

Also, if the number of concurrent updates is very high, then pessimistic locking gives more reliable results as compared to optimistic locking. This is because, with optimistic locking, there is a possibility of losing updates. Pessimistic locking happens at a database level, so there are chances of DB locking with this approach.

**On the other hand, optimistic locking is suitable for applications that do more reads than writes**. It’s favorable in situations where entities must be detached for some time and locks need not be held.

# Spring Data JPA - Applying Pessimistic Locking with @Lock Annotation

public interface ArticleRepository extends CrudRepository<Article, Long> {

@Lock(LockModeType.PESSIMISTIC\_WRITE)

@Query("select a from Article a where a.id = :id")

Article findArticleForWrite(@Param("id") Long id);

@Lock(LockModeType.PESSIMISTIC\_READ)

@Query("select a from Article a where a.id = :id")

Article findArticleForRead(@Param("id") Long id);

}

# Spring Data JPA - Optimistic Locking

@Entity

public class Employee{

private @Id

@GeneratedValue

Long id;

private String name;

private String dept;

private int salary;

@Version

private long version;

.............

}

public interface EmployeeRepository extends CrudRepository<Employee, Long> {

}

**JPA Lock Mechanism**

JPA has two main lock types defined, which are Pessimistic Locking and Optimistic Locking.

**Pessimistic Locking**

**When we are using**[**Pessimistic Locking**](https://www.baeldung.com/jpa-pessimistic-locking)**in a transaction and access an entity, it will be locked immediately. The transaction releases the lock either by committing or rolling back the transaction.**

**Optimistic Locking**

**In**[**Optimistic Locking**](https://www.baeldung.com/jpa-optimistic-locking)**, the transaction doesn't lock the entity immediately.** Instead, the transaction commonly saves the entity's state with a version number assigned to it.

When we try to update the entity's state in a different transaction, the transaction compares the saved version number with the existing version number during an update.

At this point, if the version number differs, it means that the entity can't be modified. If there is an active transaction then that transaction will be rolled back and the underlying JPA implementation will throw an *[OptimisticLockException](https://docs.oracle.com/javaee/7/api/javax/persistence/OptimisticLockException.html)*.

Apart from the version number approach, we can use other approaches such as timestamps, hash value computation, or serialized checksum, depending on which approach is the most suitable for our current development context.

To specify a lock on a custom query method of a Spring Data JPA repository, we can annotate the method with @Lock and specify the required lock mode type:

@Lock(LockModeType.OPTIMISTIC\_FORCE\_INCREMENT)

@Query("SELECT c FROM Customer c WHERE c.orgId = ?1")

**public** List<Customer> **fetchCustomersByOrgId**(Long orgId);

To enforce the lock on predefined repository methods such as findAll or findById(id), we have to declare the method within the repository and annotate the method with the Lock annotation:

@Lock(LockModeType.PESSIMISTIC\_READ)

**public** Optional<Customer> **findById**(Long customerId);

**Setting Transaction Lock Timeouts**

When using Pessimistic Locking, the database will try to lock the entity immediately. The underlying JPA implementation throws a *LockTimeoutException* when the lock cannot be obtained immediately. To avoid such exceptions, we can specify the lock timeout value.

In Spring Data JPA, the lock timeout can be specified using the *[QueryHints](https://docs.spring.io/spring-data/jpa/docs/current/api/index.html?org/springframework/data/jpa/repository/QueryHints.html)* annotation by placing a *[QueryHint](https://docs.oracle.com/javaee/7/api/javax/persistence/QueryHint.html)* on query methods:

@Lock(LockModeType.PESSIMISTIC\_READ)

@QueryHints({@QueryHint(name = "javax.persistence.lock.timeout", value = "3000")})

**public** Optional<Customer> **findById**(Long customerId);

**Lock mode types in JPA specification**

There are three Pessimistic LockModeTypes available in JPA specification.

|  |  |
| --- | --- |
| **LockModeType** | **Description** |
| PESSIMISTIC\_READ | Rows are locked and can be read by other transactions, but they cannot be deleted or modified. PESSIMISTIC\_READ guarantees repeatable reads. |
| PESSIMISTIC\_WRITE | Rows are locked and cannot be read, modified or deleted by other transactions. For PESSIMISTIC\_WRITE no phantom reads can occur and access to data must be serialized. |
| PESSIMISTIC\_FORCE\_INCREMENT | Rows are locked and cannot be modified or deleted. For versioned entities, their version number is incremented as soon as the query executes. |

It’s important to note that when locking data pessimistically, **only the data related to the particular table rows are locked (or whole table in case of PESSIMISTIC\_WRITE).** When a foreign key appears in the locked table, this relationship will be locked. **When the foreign key is in another table though, the relationship will not be locked.**

# Spring @Transactional - isolation, propagation

### [**Propagation**](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Propagation.html)

Defines how transactions relate to each other. Common options:

* REQUIRED: Code will always run in a transaction. Creates a new transaction or reuses one if available.
* REQUIRES\_NEW: Code will always run in a new transaction. Suspends the current transaction if one exists.

The default value for @Transactional is REQUIRED, and this is often what you want.

### [**Isolation**](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/transaction/TransactionDefinition.html)

Defines the data contract between transactions.

* ISOLATION\_READ\_UNCOMMITTED: Allows dirty reads.
* ISOLATION\_READ\_COMMITTED: Does not allow dirty reads.
* ISOLATION\_REPEATABLE\_READ: If a row is read twice in the same transaction, the result will always be the same.
* ISOLATION\_SERIALIZABLE: Performs all transactions in a sequence.

With a propagation level of

* REQUIRES\_NEW: we would expect fooService.provideService() was *NOT* rolled back since it created it's own sub-transaction.
* REQUIRED: we would expect everything was rolled back and the backing store was unchanged.

**PROPAGATION\_REQUIRED = 0**; If DataSourceTransactionObject T1 is already started for Method M1. If for another Method M2 Transaction object is required, no new Transaction object is created. Same object T1 is used for M2.

**PROPAGATION\_MANDATORY = 2**; method must run within a transaction. If no existing transaction is in progress, an exception will be thrown.

**PROPAGATION\_REQUIRES\_NEW = 3**; If DataSourceTransactionObject T1 is already started for Method M1 and it is in progress (executing method M1). If another method M2 start executing then T1 is suspended for the duration of method M2 with new DataSourceTransactionObject T2 for M2. M2 run within its own transaction context.

**PROPAGATION\_NOT\_SUPPORTED = 4**; If DataSourceTransactionObject T1 is already started for Method M1. If another method M2 is run concurrently. Then M2 should not run within transaction context. T1 is suspended till M2 is finished.

**PROPAGATION\_NEVER = 5**; None of the methods run in transaction context.

**An isolation level:** It is about how much a transaction may be impacted by the activities of other concurrent transactions. It a supports consistency leaving the data across many tables in a consistent state. It involves locking rows and/or tables in a database.

**The problem with multiple transaction**

**Scenario 1**. If T1 transaction reads data from table A1 that was written by another concurrent transaction T2. If on the way T2 is rollback, the data obtained by T1 is invalid one. E.g. a=2 is original data. If T1 read a=1 that was written by T2. If T2 rollback then a=1 will be rollback to a=2 in DB. But, now, T1 has a=1 but in DB table it is changed to a=2.

**Scenario2**. If T1 transaction reads data from table A1. If another concurrent transaction (T2) update data on table A1. Then the data that T1 has read is different from table A1. Because T2 has updated the data on table A1. E.g. if T1 read a=1 and T2 updated a=2. Then a!=b.

**Scenario 3**. If T1 transaction reads data from table A1 with certain number of rows. If another concurrent transaction (T2) inserts more rows on table A1. The number of rows read by T1 is different from rows on table A1.

Scenario 1 is called **Dirty reads.**

Scenario 2 is called **Non-repeatable reads.**

Scenario 3 is called **Phantom reads.**

So, isolation level is the extend to which **Scenario 1, Scenario 2, Scenario 3** can be prevented. You can obtain complete isolation level by implementing locking. That is preventing concurrent reads and writes to the same data from occurring. But it affects performance. The level of isolation depends upon application to application how much isolation is required.

**ISOLATION\_READ\_UNCOMMITTED**: Allows to read changes that haven’t yet been committed. It suffer from Scenario 1, Scenario 2, Scenario 3.

**ISOLATION\_READ\_COMMITTED**: Allows reads from concurrent transactions that have been committed. It may suffer from Scenario 2 and Scenario 3. Because other transactions may be updating the data.

**ISOLATION\_REPEATABLE\_READ**: Multiple reads of the same field will yield the same results untill it is changed by itself. It may suffer from Scenario 3. Because other transactions may be inserting the data.

**ISOLATION\_SERIALIZABLE**: Scenario 1, Scenario 2, Scenario 3 never happen. It is complete isolation. It involves full locking. It affects performace because of locking.

Enough explanation about each parameter is given by other answers; However you asked for a real world example, here is the one that clarifies the purpose of different **propagation** options:

Suppose you're in charge of implementing a *signup service* in which a confirmation e-mail is sent to the user. You come up with two service objects, one for *enrolling* the user and one for *sending* e-mails, which the latter is called inside the first one. For example something like this:

/\* Sign Up service \*/

@Service

@Transactional(Propagation=REQUIRED)

class SignUpService{

...

void SignUp(User user){

...

emailService.sendMail(User);

}

}

/\* E-Mail Service \*/

@Service

@Transactional(Propagation=REQUIRES\_NEW)

class EmailService{

...

void sendMail(User user){

try{

... // Trying to send the e-mail

}catch( Exception)

}

}

You may have noticed that the second service is of propagation type **REQUIRES\_NEW** and moreover chances are it throws an exception (SMTP server down ,invalid e-mail or other reasons).You probably don't want the whole process to roll-back, like removing the user information from database or other things; therefore you call the second service in a separate transaction.

Back to our example, this time you are concerned about the database security, so you define your DAO classes this way:

/\* User DAO \*/

@Transactional(Propagation=MANDATORY)

class UserDAO{

// some CRUD methods

}

Meaning that whenever a DAO object, and hence a potential access to db, is created, we need to reassure that the call was made from inside one of our services, implying that a live transaction should exist; otherwise an exception occurs.Therefore the propagation is of type **MANDATORY**.

**Isolation level** defines how the changes made to some data repository by one transaction affect other simultaneous concurrent transactions, and also how and when that changed data becomes available to other transactions. When we define a transaction using the Spring framework we are also able to configure in which isolation level that same transaction will be executed.

@Transactional(isolation=Isolation.READ\_COMMITTED)

public void someTransactionalMethod(Object obj) {

}

READ\_UNCOMMITTED isolation level states that a transaction may read data that is still uncommitted by other transactions.

READ\_COMMITTED isolation level states that a transaction can't read data that is not yet committed by other transactions.

REPEATABLE\_READ isolation level states that if a transaction reads one record from the database multiple times the result of all those reading operations must always be the same.

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.

**Propagation** is the ability to decide how the business methods should be encapsulated in both logical or physical transactions.

Spring REQUIRED behavior means that the same transaction will be used if there is an already opened transaction in the current bean method execution context.

REQUIRES\_NEW behavior means that a new physical transaction will always be created by the container.

The NESTED behavior makes nested Spring transactions to use the same physical transaction but sets savepoints between nested invocations so inner transactions may also rollback independently of outer transactions.

The MANDATORY behavior states that an existing opened transaction must already exist. If not an exception will be thrown by the container.

The NEVER behavior states that an existing opened transaction must not already exist. If a transaction exists an exception will be thrown by the container.

The NOT\_SUPPORTED behavior will execute outside of the scope of any transaction. If an opened transaction already exists it will be paused.

The SUPPORTS behavior will execute in the scope of a transaction if an opened transaction already exists. If there isn't an already opened transaction the method will execute anyway but in a non-transactional way.

I have run outerMethod, method\_1 and method\_2 with different propagation modes.

Below is the output for different propagation modes.

#### **Outer Method**

#### **Method\_2**

@Transactional()

@Override

public void method\_2() {

Session session = null;

try {

session = getSession();

Temp entity = new Temp(0l, "CCC");

session.save(entity);

int i = 1/0;

System.out.println("Method - 2 Id "+entity.getId());

} finally {

if (session != null && session.isOpen()) {

}

}

}

@Transactional

@Override

public void outerMethod() {

customerProfileDAO.method\_1();

iWorkflowDetailDao.method\_2();

}

#### **Method\_1**

@Transactional(propagation=Propagation.MANDATORY)

public void method\_1() {

Session session = null;

try {

session = getSession();

Temp entity = new Temp(0l, "XXX");

session.save(entity);

System.out.println("Method - 1 Id "+entity.getId());

} finally {

if (session != null && session.isOpen()) {

}

}

}

* + OuterMethod - Without transaction
  + Method\_1 - Propagation.MANDATORY) -
  + Method\_2 - Transaction annotation only
  + Output: method\_1 will throw exception that no existing transaction
  + OuterMethod - Without transaction
  + Method\_1 - Transaction annotation only
  + Method\_2 - Propagation.MANDATORY)
  + Output: method\_2 will throw exception that no existing transaction
  + Output: method\_1 will persist record in database.
  + OuterMethod - With transaction
  + Method\_1 - Transaction annotation only
  + Method\_2 - Propagation.MANDATORY)
  + Output: method\_2 will persist record in database.
  + Output: method\_1 will persist record in database. -- Here Main Outer existing transaction used for both method 1 and 2
  + OuterMethod - With transaction
  + Method\_1 - Propagation.MANDATORY)
  + Method\_2 - Transaction annotation only and throws exception
  + Output: no record persist in database means rollback done.
  + OuterMethod - With transaction
  + Method\_1 - Propagation.REQUIRES\_NEW)
  + Method\_2 - Propagation.REQUIRES\_NEW) and throws 1/0 exception
  + Output: method\_2 will throws exception so method\_2 record not persisted.
  + Output: method\_1 will persist record in database.
  + Output: There is no rollback for method\_1

**Atomicity**: It means that either a transaction happens in full or doesn’t happen at all. At any point, if the transaction feels it can’t process, it’ll rollback.

**Consistency**: It means that the state of the database remains consistent before a transaction begins and after the transaction ends.

**Isolation**: It means that multiple transactions can run in parallel without disrupting the consistency of the database.

**Durability**: It means that any changes made in the database actually persist.  
 **References**:

https://stackoverflow.com/questions/129329/optimistic-vs-pessimistic-locking

https://stackoverflow.com/questions/41303808/multiple-threads-update-same-row-in-database-at-a-time-how-to-maintain-consisten

https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/transactions.html

https://docs.jboss.org/hibernate/orm/5.2/userguide/html\_single/Hibernate\_User\_Guide.html#locking

https://blog.zestmoney.in/handling-concurrent-updates-with-optimistic-and-pessimistic-locking-in-jpa-c26d0b6855e7

https://www.logicbig.com/tutorials/spring-framework/spring-data/pessimistic-locking-and-lock-annotation.html

https://www.logicbig.com/tutorials/spring-framework/spring-data/optimistic-lock.html

https://www.baeldung.com/java-jpa-transaction-locks

https://github.com/nburana/optimisticlockdemo

https://enterprisecraftsmanship.com/posts/optimistic-locking-automatic-retry/

https://blog.mimacom.com/testing-pessimistic-locking-handling-spring-boot-jpa/

https://titanwolf.org/Network/Articles/Article?AID=8368450d-1ec8-4e3b-ab03-3ecee7ce1a3d#gsc.tab=0

https://www.baeldung.com/jpa-pessimistic-locking

https://www.baeldung.com/jpa-optimistic-locking

https://allaroundjava.com/pessimistic-locking-hibernate/

https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/pessimistic-lock.html

https://vladmihalcea.com/optimistic-vs-pessimistic-locking/

https://arnoldgalovics.com/pessimistic-locking-in-jpa-hibernate/

https://dzone.com/articles/hibernate-collections

https://vladmihalcea.com/optimistic-locking-version-property-jpa-hibernate/

https://stackoverflow.com/questions/19454003/implementing-optimistic-lock-using-hibernate-and-spring

https://www.javacodegeeks.com/2012/11/jpahibernate-version-based-optimistic-concurrency-control.html

https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html

https://stackoverflow.com/questions/8490852/spring-transactional-isolation-propagation

https://www.javainuse.com/spring/boot-transaction-isolation

https://www.baeldung.com/spring-transactional-propagation-isolation