**Optimistic Locking VS Pessimistic Locking– 2024**

**Locking is a common approach used to handle (and prevent) simultaneous access to shared data, with the intention of preventing inconsistent results.**

**@Lock is applicable only to a SELECT query**.

**Optimistic Locking**

* **Optimistic locking** is based on detecting changes on entities by checking their **version** attribute.
* In order to use optimistic locking, we need to have an entity including a property with **@Version annotation**.
* Optimistic Locking may be a better option for performance and concurrency.
* **Optimistic Locking is suitable for applications which do much more reads than updates or deletes.**
* Whenever we request the OPTIMISTIC lock mode, a persistence provider will prevent our data from dirty reads as well as non-repeatable reads

We have two different optimistic lock modes available from JPA

* **@Lock(LockModeType.*OPTIMISTIC*): lock for entities that contain a version attribute, you will get an optimistic read lock.**
* **@Lock(LockModeType.*OPTIMISTIC\_FORCE\_INCREMENT*):automatically increments the version attribute.**

**Use of Optimistic Lock using Spring JPA in SpringBoot**

**Entity Class**

@Entity(name = "Product")

@Table(name = "Product")

@Getter

@Setter

@ToString

@NoArgsConstructor

**public** **class** Product {

@Id

@GeneratedValue

**private** Long id;

@Column(name = "name")

**private** String name;

@Column(name = "price")

**private** **int** price;

@Version

**private** Integer version;

**public** Product(String name, **int** price) {

**this**.name = name;

**this**.price = price;

}

}

Repository Layer

@Repository

**public** **interface** ProductRepository **extends** CrudRepository<Product, Long> {

@Lock(LockModeType.***OPTIMISTIC***)

Optional<Product> findProductById(Long Id);

// Useful to find popular product

@Lock(LockModeType.***OPTIMISTIC\_FORCE\_INCREMENT***)

@Query("select p from Product p where p.name=:name")

Optional<Product> getproductByName(@Param("name") String name);

// Useful to find popular product

@Lock(LockModeType.***OPTIMISTIC\_FORCE\_INCREMENT***)

@Query("select p from Product p where p.name=:name")

Optional<Product> searchProductByName(@Param("name") String name);

@Modifying

@Query("update Product p set p.price = :price where p.id = :id")

**void** updatePrice(@Param("price") **int** price, @Param("id") **long** id);

}

Service Layer

**public** **interface** ProductService {

Product saveProduct(Product p);

**void** saveAllProducts(List<Product> ps);

Product getProductById(**long** id);

**void** updatePrice(Product p);

Product getProductByName(String name);

}

Service Implementation Class

@Service

**public** **class** ProductServiceImpl **implements** ProductService {

@Autowired

**private** ProductRepository productRepo;

@Transactional

@Override

**public** Product saveProduct(Product p) {

**return** Optional.*ofNullable*(productRepo.save(p)).get();

}

@Transactional

@Override

**public** **void** saveAllProducts(List<Product> ps) {

productRepo.saveAll(ps);

}

@Transactional

@Override

**public** Product getProductById(**long** id) {

Product prod = **null**;

Optional<Product> optProd = productRepo.findProductById(id);

**if**(!optProd.isEmpty())

prod = optProd.get();

**return** prod;

}

@Transactional

@Override

**public** **void** updatePrice(Product p) {

productRepo.updatePrice(p.getPrice(), p.getId());

}

@Transactional

@Override

**public** Product getProductByName(String name) {

Product prod = **null**;

Optional<Product> optProd = productRepo.getproductByName(name);

**if**(!optProd.isEmpty())

prod = optProd.get();

**return** prod;

}

}

AutoRun class

@Component

**public** **class** AutoRun {

@Autowired

**private** ProductService prodService;

**private** **void** saveProducts() {

List<Product> prodList = List.*of*(**new** Product("Samsung", 2000),

**new** Product("Nokia", 3000), **new** Product("HTC Mobile", 4000));

prodService.saveAllProducts(prodList);

}

**public** Product getProductById(**long** productId) {

Product prod = prodService.getProductById(productId);

System.***out***.println("Product Details: "+prod);

**return** prod;

}

**public** **void** getProductByName(String name) {

Product prod = prodService.getProductByName(name);

System.***out***.println("Product Details with name: "+prod);

}

@EventListener(ApplicationReadyEvent.**class**)

**public** **void** checkAndRun() {

System.***out***.println("Application Stated Running ....");

// saveProducts();

//Get product by id and no need to increment version

// Product prod = getProductById(102);

// prod.setPrice(6000);

// prodService.updatePrice(prod);

// System.out.println("Get the updated product again ...");

// getProductById(102);

// Get product by name and increment version

getProductByName("Nokia");

}

}

Note: You can use @Lock(LockModeType.***OPTIMISTIC\_FORCE\_INCREMENT***) for a use that if the product name has more higher version, it means people are searching for this product. It can also be used for searching the product. You can use @Lock(LockModeType.***OPTIMISTIC***) for normal usage, it means a page simply displays all products and user is opening randomly.

**Application.properties**

server.port=9090

## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)

spring.datasource.url=jdbc:postgresql://localhost:5432/postgres

spring.datasource.username=postgres

spring.datasource.password=postgres

# The SQL dialect makes Hibernate generate better SQL for the chosen database

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect

# Hibernate ddl auto (create, create-drop, validate, update)

spring.jpa.hibernate.ddl-auto = update

#spring.jpa.properties.hibernate.generate\_statistics=true

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format\_sql=true

spring.jpa.properties.hibernate.temp.use\_jdbc\_metadata\_defaults=false

**Pessimistic Locking**

* **Pessimistic Locking is when you lock the entire 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.
* Pessimistic locking is used when a collision is anticipated. the pessimistic locking mechanism involves locking entities on the database level.
* As long as it holds the lock, no transaction can read, delete or make any updates on the locked data.
* We can use a pessimistic lock to ensure that **no other transactions can modify or delete reserved data**.
* **There are two types of locks** we can retain: **an exclusive lock** and **a shared lock**.
* We could read but not write in data when someone else holds a shared lock.
* In order to modify or delete the reserved data, we need to have an exclusive lock.
* We can presume that using pessimistic locking may result in deadlocks. However, it ensures greater integrity of data than optimistic locking.
* The pessimistic locking method forbids concurrent record updates
* **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.
* **Pessimistic locking is also more appropriate in applications that contain small tables that are frequently updated.**

**Pessimistic Lock Modes**

Same as optimistic locking, JPA defines pessimistic lock modes in the LockModeType class:

**@Lock(LockModeType.PESSIMISTIC\_READ): allows us to obtain a shared lock and prevents data from being modified. Whenever we want to just read data and don’t encounter dirty reads, we could use PESSIMISTIC\_READ (shared lock).** Whenever **we want to just read data and don't encounter dirty reads**, we could use **PESSIMISTIC\_READ (shared lock)**. We won't be able to make any updates or deletes though. You obtain a lock on the record at the start of the transaction, for the purpose of reading only. Basically you're saying "I don't want anyone updating this record while I'm reading it, but I don't mind others reading it as well". That means people also attempting a PESSIMISTIC\_READ will succeed, but those attempting a PESSIMISTIC\_WRITE will fail.

**@Lock(LockModeType.PESSIMISTIC\_WRITE): allows us to obtain an exclusive lock and prevents other transactions from reading or modifying the data.** According to the JPA specification, **holding PESSIMISTIC\_WRITE lock will prevent other transactions from reading, updating or deleting the data**. you obtain a lock on the record at the start of the transaction, for the purpose of writing. What you're saying is "I'm going to be updating this record, so no one can read or write to it until I'm done".

**@Lock(LockModeType.PERRIMISTIC\_FORCE\_INCREMENT): it additionally increments a version attribute of a versioned entity.**

In summary, Pessimistic Locking may be a better choice for data integrity and consistency; however, in large-scale applications, it can negatively impact performance by reducing concurrency. On the other hand, Optimistic Locking may be a better option for performance and concurrency; however, caution is needed in handling conflicts. It’s important to note that these choices may vary based on application requirements, performance expectations, and data integrity needs.

Example is given below.

Entity Layer

@Entity(name = "Item")

@Table(name = "Item")

@Getter

@Setter

@ToString

@NoArgsConstructor

**public** **class** Item {

@Id

@GeneratedValue

**private** Long id;

@Column(name = "name")

**private** String name;

@Column(name = "rating")

**private** **int** rating;

@Column(name = "price")

**private** **int** price;

@Version

**private** Integer version;

**public** Item(String name, **int** rating, **int** price) {

**super**();

**this**.name = name;

**this**.rating = rating;

**this**.price = price;

}

}

Repository Layer

@Repository

**public** **interface** ItemRepository **extends** CrudRepository<Item, Long> {

@Lock(LockModeType.***PESSIMISTIC\_READ***)

@Query("select i from Item i where i.id = :id")

Item findItemForRead(@Param("id") Long id);

@Lock(LockModeType.***PESSIMISTIC\_WRITE***)

@Query("select i from Item i where i.id = :id")

Item findItemForWrite(@Param("id") Long id);

@Lock(LockModeType.***PESSIMISTIC\_FORCE\_INCREMENT***)

@Query("select i from Item i where i.name = :name")

Item updateItemForRating(@Param("name") String name);

@Lock(LockModeType.***PESSIMISTIC\_FORCE\_INCREMENT***)

@Query("select i from Item i where i.name = :name")

Item setRatingForItem(@Param("name") String name);

}

Service Layer

**public** **interface** ItemService {

Item getItemById(Long id);

Item saveItem(Item item);

**void** saveAllItems(List<? **extends** Item> items);

**void** updateItemPrice(**long** id, **int** price);

**void** updateItemRating(String name, **int** rating);

**void** setItemRating(String name, **int** rating);

}

Service Implementation class

@Service

**public** **class** ItemServiceImpl **implements** ItemService {

@Autowired

**private** ItemRepository itemRepo;

@Transactional

@Override

**public** Item getItemById(Long id) {

**return** itemRepo.findItemForRead(id);

}

@Transactional

@Override

**public** Item saveItem(Item item) {

**return** itemRepo.save(item);

}

@Transactional

@Override

**public** **void** saveAllItems(List<? **extends** Item> items) {

itemRepo.saveAll(items);

}

@Transactional

@Override

**public** **void** updateItemPrice(**long** id, **int** price) {

Item item = itemRepo.findItemForWrite(id);

item.setPrice(price);

itemRepo.save(item);

}

@Transactional

@Override

**public** **void** updateItemRating(String name, **int** rating) {

//It will update automatically

itemRepo.updateItemForRating(name);

}

@Transactional

@Override

**public** **void** setItemRating(String name, **int** rating) {

Item item = itemRepo.setRatingForItem(name);

System.***out***.println("Now Mobile Item: "+item);

}

}

Test Method with AutoRun class

**public** **void** checkForPessimistic() {

List<Item> items = List.*of*(**new** Item("Mobile", 3, 3000), **new** Item("Office NoteBook", 4, 1000),

**new** Item("Electric Bike", 2, 27000), **new** Item("Medicine", 5, 100), **new** Item("Cooktop", 1, 2000),

**new** Item("Grocery", 5, 1000));

itemService.saveAllItems(items);

// Only for reading, office notebook

Item item1 = itemService.getItemById(Long.*valueOf*(103));

System.***out***.println("Item1 details: "+item1);

// Only for writing price, do not update version for cooktop

itemService.updateItemPrice(106, 3000);

// Get the updated item price

Item item2 = itemService.getItemById(Long.*valueOf*(106));

System.***out***.println("Item2 details: "+item2);

//

// //Based upon mobile usage, rating is updated for mobile

itemService.updateItemRating("Mobile",5);

Item item3 = itemService.getItemById(Long.*valueOf*(102));

System.***out***.println("Item3 details: "+item3);

// Check for Medicine

Item item5 = itemService.getItemById(Long.*valueOf*(105));

System.***out***.println("Item1 details: "+item5);

itemService.setItemRating("Mobile",4);

Item item7 = itemService.getItemById(Long.*valueOf*(102));

System.***out***.println("Item3 details: "+item7);

}

@EventListener(ApplicationReadyEvent.**class**)

**public** **void** checkAndRun() {

System.***out***.println("Application Stated Running ....");

checkForPessimistic();

}

**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 annotation** which defines a single attribute of type org.hibernate.annotations.**OptimisticLockType**. 🡺 Hibernate specific.

Here are **4 available OptimisticLockTypes**:

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

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

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

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

@Entity(name = "Person")

**@OptimisticLocking(type = OptimisticLockType.ALL)**

**@DynamicUpdate**

public static class Person {

@Id

private Long id;

@Column(name = "`name`")

private String name;

//Getters and setters are omitted for brevity

}

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

**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;

//Getters and setters are omitted for brevity

}

**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 **@SelectBeforeUpdate annotation** so that detached entities are properly handled by the Session#update(entity) operation**.**

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 ;

**@Version**

**private long version;**

...

}

**Multiple threads update same row in database at a time how to maintain consistency? 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).

**Setting Transaction Lock Timeouts**

**@Lock(LockModeType.PESSIMISTIC\_READ)**

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

public Optional<Customer> findById(Long customerId);

**@Transactional and @Lock (Why are we using @Lock when we already have @Transactional?)**

**Transactional:** Whenever you put @Transactional annotation, it enables transactional behavior which qualifies ACID properties. You need to start a transaction before executing a locking query. You can only lock something in the context of a database transaction.

**Atomic** Guarantees that all operations in a transaction are treated as a single “unit”, which either succeeds completely or fails completely.

**Consistent** Ensures that a transaction can only bring the database from one valid state to another by preventing data corruption.

**Isolation** Determines how and when changes made by one transaction become visible to the other. Serializable and Snapshot Isolation are the top 2 isolation levels from a strictness standpoint.

**Durable** Ensures that the results of the transaction are permanently stored in the system. The modifications must persist even in case of power loss or system failures.

**Lock:** It should not be confused with transactional, @Lock enables locking behavior during a **transaction.**

@Lock only works for that repository call and not the whole @Transactional method call

**If you want transactional behavior then add @transactional and if your usecase requires locking and as per use case use appropriate locking.**

**The two main tools we use to cope with concurrency are database transactions and distributed locks. These two are not interchangeable. You can't use a transaction when you need a lock. You can't use a lock when you need a transaction.** [source](https://makandracards.com/makandra/31937-differences-between-transactions-and-locking).

If you look carefully at the javadoc of @Lock it states that: Annotation used to specify the LockModeType to be used when executing the query. So the lock is active only during the query execution.. after the query is done.. the lock is released. It is not active for the duration of the entire transaction as you expect it to be.

Why are we using @Lock when we already have @Transactional?

<https://stackoverflow.com/questions/58786195/what-is-the-relation-between-spring-transactional-and-spring-lock-annotation>

<https://makandracards.com/makandra/31937-differences-between-transactions-and-locking>

When your code needs to deal with concurrent data access, the two main tools we use to cope with concurrency are **database transactions** and **distributed locks**. These two are **not** interchangeable.

Use a **transaction** to ensure a list of changes is applied **either completely or not at all**.

**When should I use a transaction?**

Whenever you make an operation that affects more than a single database row, you should **always** use a transaction.

**What happens if two transactions run at the same time?**

When two concurrently running transactions are done, they will each try to commit their changes in one, atomic commit. This may or may not succeed. E.g. if both transactions try to insert the same unique key, whatever transaction commits first will succeed. The other transaction will get an error from the database adapter and all of its changes will be rolled back.

**Use locks to prevent concurrent data access**

Use **locks** to ensure that a critical piece of code only runs in a single thread at the same time. A lock is also called a "mutex".