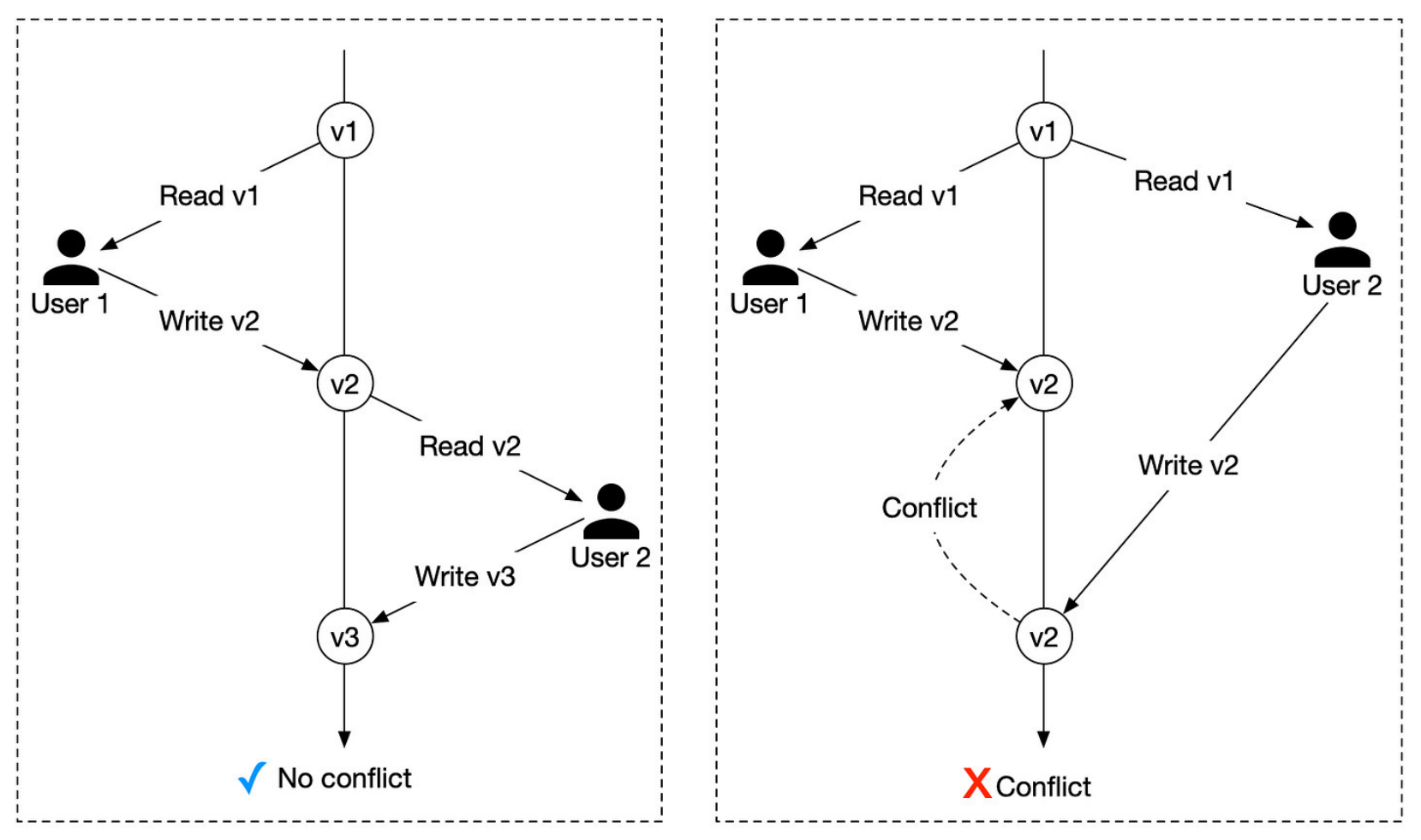
Transactions

Transactional Locks in Spring Data JPA

Conflicts in Concurrent Transactions



Transaction Locks

Transaction locks in Spring Data JPA are a mechanism used to ensure data consistency in multi-threaded environments where multiple transactions may attempt to access and modify the same data concurrently.

Spring Data JPA offers transactional locks that allow developers to lock a specific entity or set of entities during a transaction. When an entity is locked, it can only be accessed and modified by the transaction that holds the lock, and other transactions are prevented from accessing or modifying the entity until the lock is released.

Why use Transaction Locks

When we already have Isolation Levels which Define the visibility and consistency guarantees provided by the database for concurrent transactions.

Transaction Locks implement the mechanisms needed to enforce these guarantees and prevent conflicts by controlling access to data.

Therefore Locks are necessary to ensure that the guarantees provided by isolation levels are practically enforced and to handle complex concurrency scenarios effectively.

In essence, isolation levels set the rules for transaction behavior, while locks are the tools that implement these rules in practice.

Optimistic Locking

1. A new column called “version” is added to the database table.

2. Before a user modifies a database row, the application reads the version number of the row.

3. When the user updates the row, the application increases the version number by 1 and writes it back to the database.

4. A database validation check is put in place; the next version number should exceed the current version number by 1. The transaction aborts if the validation fails and the user tries again from step 2.

Optimistic Locking

1. A new column called “version” is added to the database table.

2. Before a user modifies a database row, the application reads the version number of the row.

3. When the user updates the row, the application increases the version number by 1 and writes it back to the database.

4. A database validation check is put in place; the next version number should exceed the current version number by 1. The transaction aborts if the validation fails and the user tries again from step 2.

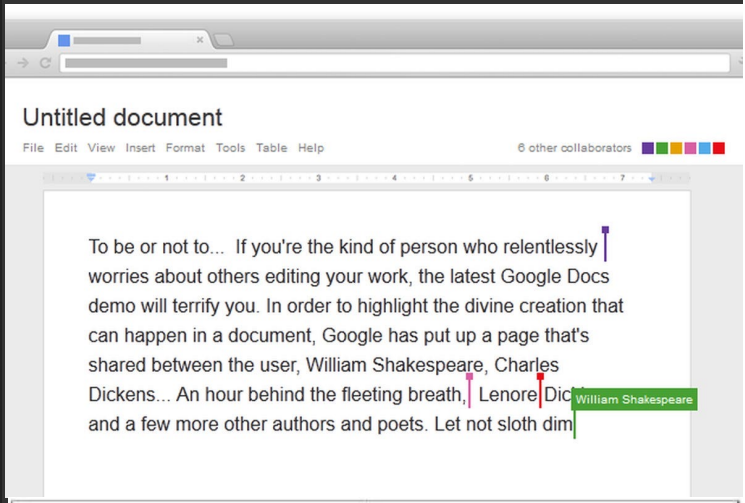
Pessimistic Locking

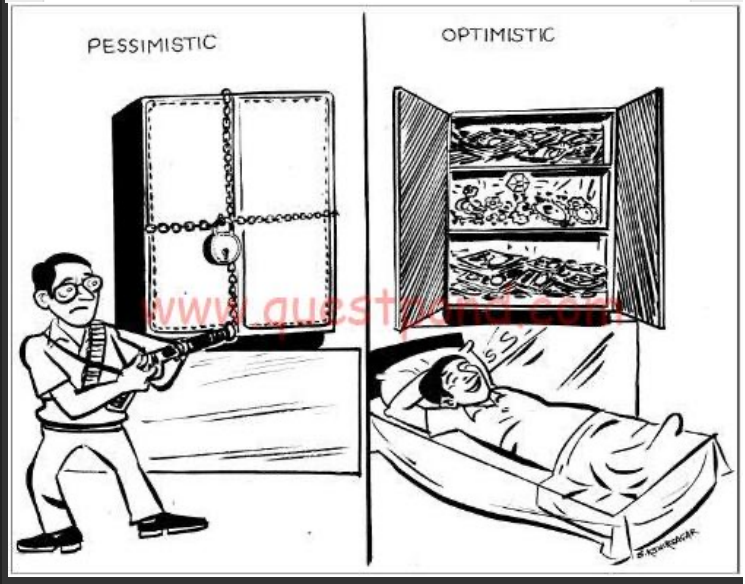
Use the @Lock annotation in your repository interface to specify the lock mode

- A PESSIMISTIC_READ lock allows you to read the data while ensuring that no other transaction can modify the data until the current transaction completes. This is also known as a shared lock.

- A PESSIMISTIC_WRITE lock (also known as an exclusive lock) prevents both reads and writes by other transactions on the locked row. This ensures that only the transaction holding the lock can modify the row.

Feature	Optimistic Locking	Pessimistic Locking
Concurrency	Suitable for high-concurrency environments; assumes conflicts are rare.	Better suited for low-concurrency environments or when conflicts are likely.
Performance	No locking at the database level, so it's more efficient with less contention.	Requires database-level locking, which can decrease performance due to lock acquisition and release overhead.
Failure Handling	Updates fail only if the `version` has changed, making it more efficient.	Other transactions are blocked until the lock is released, which can lead to longer wait times or deadlocks.
Lock Type	No explicit database lock; relies on a version check.	Explicitly locks the row or table in the database until the transaction completes.
Lock Scope	Transaction fails only when a conflicting update occurs.	Prevents other transactions from modifying or even reading the locked data.
Use Case	Best for scenarios where conflicts are rare but possible (e.g., read-heavy systems).	Best for scenarios where conflicts are common and you need to prevent data changes (e.g., financial transactions).
Drawbacks	Conflicts are detected late, possibly requiring retries.	Can lead to deadlocks, slower performance due to locks, and longer transaction times.





Transaction Strategies in real world

- **Read Committed**: E-commerce, online transactions, where you need consistent data but prioritize performance (e.g., Amazon).
- **Read Uncommitted**: Data analytics, non-critical reporting (e.g., Google Analytics).
- **Serializable**: Financial transactions, stock trading, critical bank operations (e.g., SBI, PNB).
- **Optimistic Locking**: Collaborative editing, where conflicts are rare but need to be managed without locking (e.g., Google Docs).
- **Pessimistic Locking**: Resource booking or systems requiring exclusive access to avoid conflicts (e.g., Booking.com, airline reservations).
- **No Transactions**: Social media, large-scale distributed systems, where eventual consistency is acceptable (e.g., Facebook, Twitter).