Transactions & Atomicity

Tuesday, October 28, 2025 2:09 AM

1️⃣ Transactions in MongoDB

MongoDB supports **multi-document ACID transactions** since **v4.0** (for replica sets) and **v4.2** (for sharded clusters).

Key Points:

- **Single-document operations** are always atomic.
- **Multi-document transactions** ensure **ACID guarantees**: Atomicity, Consistency, Isolation, Durability.
- Transactions are created using **sessions** in MongoDB drivers:

Example (Java/MongoDB):

```
ClientSession session = mongoClient.startSession();
try {
    session.startTransaction();

    collection1.insertOne(session, doc1);
    collection2.updateOne(session, filter, update);

    session.commitTransaction();
} catch (Exception e) {
    session.abortTransaction();
} finally {
    session.close();
}
```

- **Isolation:** MongoDB transactions provide **snapshot isolation**, meaning each transaction sees a consistent snapshot of the data at its start.

2️⃣ Handling Concurrent Requests (Two Users Hitting Same Request)

MongoDB uses **document-level concurrency**:

- Only one operation can modify a single document at a time.
- If two users try to update the **same document**, the **first write wins**, the second write may fail if you rely on conditional updates (updateOne with a filter on a version or timestamp).

Patterns for handling concurrency:

a) Optimistic Concurrency Control (OCC)

- Add a version field to the document.
- Update only if version matches:

```
collection.updateOne(
    Filters.and(Filters.eq("_id", docId), Filters.eq("version",
currentVersion)),
    Updates.combine(
        Updates.set("field", newValue),
        Updates.inc("version", 1)
    )
);
```

- `If update count = 0 → retry or abort.`

b) Pessimistic Locking (Limited in MongoDB)

- MongoDB **does not support row-level locks like SQL**.
- You can emulate locks with a **"lock" document**:

{ "_id": "resource_lock", "lockedBy": "instance1", "timestamp": 12345678 }

- Use findOneAndUpdate with a filter to acquire lock atomically.
- Release after operation.

3️⃣ Multiple Instances Writing to DB

When **multiple app instances** write:

- MongoDB **replica sets handle replication**; writes go to the primary.
- To prevent **race conditions**, use **atomic operations** ($set, $inc, $push, findOneAndUpdate).
- For **distributed locks**, you can implement a **lock collection** (like Redis locks) or use MongoDB's findOneAndUpdate with a TTL index for auto-expiry.

4️⃣ Comparison with Spring Transactions + ShedLock

Aspect	MongoDB	Spring + RDB + ShedLock
Transaction Scope	Session-based; multi-document ACID	Method-level @Transactional

Isolation	Snapshot isolation	Read Committed / Repeatable Read / Serializable
Concurrent Writes	Use OCC / locks at document level	Database handles row-level locks; @Transactional ensures atomicity
Distributed Jobs / Locks	Implement manual lock collection or TTL-based lock	ShedLock library ensures only **one node runs a scheduled job** in a cluster
Locking Mechanism	Atomic updates, optionally explicit locks	Database row-level locking or explicit ShedLock table row

Key takeaway:

- **MongoDB relies more on atomic operations and optimistic concurrency**, while **Spring + RDB** relies on DB locks and transaction isolation.
- For distributed systems with multiple nodes, **ShedLock** is a neat abstraction; in MongoDB, you implement a **lock collection** or use Redis/Zookeeper to prevent double execution.

5️⃣ Example: Distributed Lock in MongoDB

```
Document lockDoc = new Document("_id", "job1")
                     .append("lockedAt", Instant.now())
                     .append("lockedBy", instanceId);

Document result = lockCollection.findOneAndUpdate(
    Filters.eq("_id", "job1"),
    new Document("$setOnInsert", lockDoc),
    new
FindOneAndUpdateOptions().upsert(true).returnDocument(ReturnDocument.A
FTER)
);

if (result.getString("lockedBy").equals(instanceId)) {
    // acquired lock, safe to run job
}
```

- **$setOnInsert + upsert** ensures only one instance acquires lock.
- Release lock after job completes by deleting the document.

✅ Summary / Practical Advice

1. **MongoDB transactions** are ACID but scoped by session; prefer **atomic single-document updates** where possible.
2. **Concurrent writes** → use **optimistic concurrency** or atomic operations.
3. **Multiple instances** → handle with **atomic updates** + **lock collection** for distributed locks.
4. **Spring analogy**: transactions are like Mongo sessions; ShedLock is like a distributed lock collection in MongoDB.