Spring Framework 7

Beginner to Guru

Database Transactions

SQL Database Transactions - ACID

- **ACID**
  - **Atomicity** - All operations are completed successfully or database is returned to previous state.
  - **Consistency** - Operations do not violate system integrity constraints.
  - **Isolated** - Results are independent of concurrent transactions.
  - **Durable** - Results are made persistent in case of system failure (ie written to disk)

Important Terms

- **Transaction** - A unit of work. One or more SQL operations
  - Typically DML (and **Not** DDL) statements which alter data.
  - Can be just one; can be hundreds or thousands.
- **Commit** - Indicates the end of the transaction and tells database to make changes permanent.
  - More efficient to do multiple operations in a transaction. There is a 'cost' with commits.
- **Rollback** - Revert all changes of the transaction
- **Save Point** - Programatic point you can set, which allows you to rollback to (ie rollback part of a transaction)

Database Locks

- The database will lock the records, (in some cases whole table or database) to prevent other processes from changing data
  - ACID compliance
- Within a transaction the following DML statements will lock records of the affected rows:
  - `SELECT FOR UPDATE; UPDATE; DELETE`
- During the transactions other sessions attempting to modify locked records will by default wait for the lock to be released. (ie interactively it will seem like things are hanging)
- Deadlock - Occurs where two transactions lock each other and can never complete.
  - Both fail and roll back.

Transaction Isolation Levels

- **Repeatable Read** - Default Isolation Level. Your statement receives a consistent view of the database, even if other transactions are committed during your transaction.
  - Your transaction gets a snapshot of the data, which does not change.
- **Read Committed** - Reads within your transaction will receive a fresh snapshot of the data.
- **Read Uncommitted** - Reads are not consistent, but may avoid additional database locks.
  - aka - “Dirty Read”
- **Serializable** - Similar to Repeatable Read, but may lock rows selected in transaction.

Pragmatic Concepts to Remember

- Using the default transaction isolation level, your transaction see's a snapshot of the database as it is at the start of the transaction.
  - Changes made in other sessions and committed **WILL NOT** be visible
  - Changes made by your session **WILL NOT** be visible to other sessions until commit
- Most modern RDBMS do a good job of ACID compliance
  - Support for ACID with NoSQL database varies widely by vendor
  - ACID compliance is complex and costly - hence high performance of NoSQL databases

The “Lost” Update

- Database record has quantity of 10
- **Session A** reads 10, adds 5 making quantity 15, database record is Locked during update
- **Session B** reads 10, adds 5 making quantity 15, but is blocked by the lock of **Session A**
  - Would be 20, if **Session B** could see the uncommitted change
- **Session A** commits record, releasing lock. Database record is updated to 15
- **Session B** released, updates database record to 15
  - Thus the update of **Session A** is “Lost”

JDBC Locking Modes

- JDBC Drivers support several different locking modes
- Mode applies to lifespan of the connection
- Configuration is **very** vendor dependent
- Rarely used in practice.
- JPA/Hibernate is generally favored

JPA Locking

- **Pessimistic Locking**
  - Database mechanisms are used to lock records for updates
  - Capabilities vary widely depending on database and version of JDBC driver used
  - Simplest version is “SELECT FOR UPDATE…” - Locks row or rows until commit or rollback is issued
- **Optimistic Locking**
  - Done by checking a version attribute of the entity

JPA Locking - Which to Use???

- **Do You Need Locking**?
  - Will your application have concurrent updates of the same records???
- **Pessimistic Locking**
  - Use if the data is frequently updated, and if updated concurrently
  - Remember, there is a cost to performing the locking
- **Optimistic Locking**
  - Use if the data is read more often than updated
  - Majority of applications will use Optimistic Locking

Multi-Request Conversations

- **Multi-Request Conversation** - Occurs in web form applications, or possibly RESTful too, where the update logic is over one or more requests, thus leaving a larger window of time.
  - Pessimistic Locking is very fast, milliseconds. Will only protect against conflicts at write time.
  - Optimistic Locking provides a mechanism to detect stale data over a longer period of time (ie - multiple requests)

JPA Pessimistic Locking

- **Pessimistic Lock Modes**
  - **PESSIMISTIC_READ** - uses shared lock, prevents data from being updated or deleted
  - **PESSIMISTIC_WRITE** - uses exclusive lock, prevents data from being read (in some isolation levels), updated or deleted
  - **PESSIMISTIC_FORCE_INCREMENT** - uses exclusive lock, increments version property of entity
- Most databases will support **PESSIMISTIC_WRITE**, this is the option you will typically use
  - Use **PESSIMISTIC_FORCE_INCREMENT** if entity has version property

JPA Optimistic Locking

- Uses a version property, which is incremented with each update
- Can be int, Integer, long, Long, short, Short, or java.sql.Timestamp
  - Most common to use is Integer
- Prior to an update, Hibernate will read the corresponding database record. If the version does not match, an exception is thrown
- **Downsides**:
  - Updates outside of JPA/Hibernate which do not update the version property will break this
  - Performance - read before each update

JPA Optimistic Lock Modes

- **OPTIMISTIC** - Obtains optimistic read lock for all entities with version attribute
- **OPTIMISTIC_FORCE_INCREMENT** - Same at OPTIMISTIC, but in increments the version value
- **READ** - JPA 1.x, same as OPTIMISTIC
- **WRITE** - JPA 1.x, same as OPTIMISTIC_FORCE_INCREMENT

SPRING FRAMEWORK

GURU