10. LockingIn a relational database, locking refers to actions taken to prevent data from changing between the time it is read and the time is used. Your locking strategy can be either optimistic or pessimistic. 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. Hibernate provides mechanisms for implementing both types of locking in your applications. 10.1. 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. A version or timestamp property can never be null for a detached instance. Hibernate detects any instance with a null version or timestamp as transient, regardless of other unsaved-value strategies that you specify. Declaring a nullable version or timestamp property is an easy way to avoid problems with transitive reattachment in Hibernate, especially useful if you use assigned identifiers or composite keys. 10.1.1. Mapping optimistic locking Jakarta Persistence 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 jakarta.persistence.Version to the persistent attribute that defines the optimistic locking value. According to Jakarta Persistence, the valid types for these attributes are limited to: int or Integer short or Short long or Long java.sql.Timestamp However, Hibernate allows you to use even Java 8 Date/Time types, such as Instant. Example 434. @Version annotation mapping @Entity(name = "Person")
public static class Person {
@Id
@GeneratedValue
private Long id;
@Column(name = "`name`")
private String name;
@Version
private long version;
//Getters and setters are omitted for brevity
} @Entity(name = "Person")
public static class Person {
@Id
@GeneratedValue
private Long id;
@Column(name = "`name`")
private String name;
@Version
private Timestamp version;
//Getters and setters are omitted for brevity
} @Entity(name = "Person")
public static class Person {
@Id
@GeneratedValue
private Long id;
@Column(name = "`name`")
private String name;
@Version
private Instant version;
//Getters and setters are omitted for brevity
} Dedicated version number The version number mechanism for optimistic locking is provided through a @Version annotation. Example 435. @Version annotation @Version
private long version; Here, the version property is mapped to the version column, and the entity manager uses it to detect conflicting updates, and prevent the loss of updates that would otherwise be overwritten by a last-commit-wins strategy. The version column can be any kind of type, as long as you define and implement the appropriate UserVersionType. Your application is forbidden from altering the version number set by Hibernate. To artificially increase the version number, see the documentation for properties LockModeType.OPTIMISTIC\_FORCE\_INCREMENT or LockModeType.PESSIMISTIC\_FORCE\_INCREMENT check in the Hibernate Entity Manager reference documentation. If the version number is generated by the database, such as a trigger, use the annotation @org.hibernate.annotations.Generated(GenerationTime.ALWAYS) on the version attribute. Timestamp 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 property type. Example 436. Using timestamps for optimistic locking @Version
private Date version; The timestamp can also be generated by the database, instead of by the VM, using the @CurrentTimestamp annotation, or even @Generated(value = ALWAYS, sql = "current\_timestamp"). Example 437. Database-generated version timestamp mapping @Entity(name = "Person")
public static class Person {
@Id
private Long id;
private String firstName;
private String lastName;
@Version @CurrentTimestamp
private LocalDateTime version; Now, when persisting a Person entity, Hibernate calls the database-specific current timestamp retrieval function: Example 438. Database-generated version timestamp example Person person = new Person();
person.setId(1L);
person.setFirstName("John");
person.setLastName("Doe");
assertNull(person.getVersion());
entityManager.persist(person);
assertNotNull(person.getVersion()); CALL current\_timestamp()
INSERT INTO
Person
(firstName, lastName, version, id)
VALUES
(?, ?, ?, ?)
-- binding parameter [1] as [VARCHAR] - [John]
-- binding parameter [2] as [VARCHAR] - [Doe]
-- binding parameter [3] as [TIMESTAMP] - [2017-05-18 12:03:03.808]
-- binding parameter [4] as [BIGINT] - [1] Excluding attributes By default, every entity attribute modification is going to trigger a version incrementation. If there is an entity property which should not bump up the entity version, then you need to annotate it with the Hibernate @OptimisticLock annotation, as illustrated in the following example. Example 439. @OptimisticLock mapping example @Entity(name = "Phone")
public static class Phone {
@Id
private Long id;
@Column(name = "`number`")
private String number;
@OptimisticLock(excluded = true)
private long callCount;
@Version
private Long version;
//Getters and setters are omitted for brevity
public void incrementCallCount() {
this.callCount++;
}
} This way, if one thread modifies the Phone number while a second thread increments the callCount attribute, the two concurrent transactions are not going to conflict as illustrated by the following example. Example 440. @OptimisticLock exlude attribute example doInJPA(this::entityManagerFactory, entityManager -> {
Phone phone = entityManager.find(Phone.class, 1L);
phone.setNumber("+123-456-7890");
doInJPA(this::entityManagerFactory, \_entityManager -> {
Phone \_phone = \_entityManager.find(Phone.class, 1L);
\_phone.incrementCallCount();
log.info("Bob changes the Phone call count");
});
log.info("Alice changes the Phone number");
}); -- Bob changes the Phone call count
update
Phone
set
callCount = 1,
"number" = '123-456-7890',
version = 0
where
id = 1
and version = 0
-- Alice changes the Phone number
update
Phone
set
callCount = 0,
"number" = '+123-456-7890',
version = 1
where
id = 1
and version = 0 When Bob changes the Phone entity callCount, the entity version is not bumped up. That’s why Alice’s UPDATE succeeds since the entity version is still 0, even if Bob has changed the record since Alice loaded it. Although there is no conflict between Bob and Alice, Alice’s UPDATE overrides Bob’s change to the callCount attribute. For this reason, you should only use this feature if you can accommodate lost updates on the excluded entity properties. 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 also useful for use with modeling legacy schemas. The idea is that you can get Hibernate to perform "version checks" using either all of the entity’s attributes or just the attributes that have changed. This is achieved through the use of the @OptimisticLocking annotation which defines a single attribute of type org.hibernate.annotations.OptimisticLockType. There 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 Example 441. OptimisticLockType.ALL mapping example @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
} When you need to modify the Person entity above: Example 442. OptimisticLockType.ALL update example 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] 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 OptimisticEntityLockException 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. 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. Example 443. OptimisticLockType.DIRTY mapping example @Entity(name = "Person")
@OptimisticLocking(type = OptimisticLockType.DIRTY)
@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
} When you need to modify the Person entity above: Example 444. OptimisticLockType.DIRTY update example 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] This time, only the database column that has changed was used in the WHERE clause. 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 OptimisticEntityLockException 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) operation. 10.2. 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. Hibernate always uses the locking mechanism of the database, and never lock objects in memory. 10.3. LockMode and LockModeType Long before Java Persistence 1.0, Hibernate already defined various explicit locking strategies through its LockMode enumeration. Jakarta Persistence comes with its own LockModeType enumeration which defines similar strategies as the Hibernate-native LockMode. LockModeType LockMode Description NONE 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. READ and OPTIMISTIC READ The entity version is checked towards the end of the currently running transaction. WRITE and OPTIMISTIC\_FORCE\_INCREMENT WRITE The entity version is incremented automatically even if the entity has not changed. PESSIMISTIC\_FORCE\_INCREMENT PESSIMISTIC\_FORCE\_INCREMENT The entity is locked pessimistically and its version is incremented automatically even if the entity has not changed. PESSIMISTIC\_READ PESSIMISTIC\_READ The entity is locked pessimistically using a shared lock if the database supports such a feature. Otherwise, an explicit lock is used. PESSIMISTIC\_WRITE PESSIMISTIC\_WRITE, UPGRADE The entity is locked using an explicit lock. PESSIMISTIC\_WRITE with a jakarta.persistence.lock.timeout setting of 0 UPGRADE\_NOWAIT The lock acquisition request fails fast if the row s already locked. PESSIMISTIC\_WRITE with a jakarta.persistence.lock.timeout setting of -2 UPGRADE\_SKIPLOCKED The lock acquisition request skips the already locked rows. It uses a SELECT … FOR UPDATE SKIP LOCKED in Oracle and PostgreSQL 9.5, or SELECT … with (rowlock, updlock, readpast) in SQL Server. 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(). If you call Session.load() with option UPGRADE, UPGRADE\_NOWAIT or UPGRADE\_SKIPLOCKED, and the requested object is not already loaded by the session, the object is loaded using SELECT … FOR UPDATE. If you call load() for an object that is already loaded with a less restrictive lock than the one you request, Hibernate calls lock() for that object. Session.lock() performs a version number check if the specified lock mode is READ, UPGRADE, UPGRADE\_NOWAIT or UPGRADE\_SKIPLOCKED. In the case of UPGRADE, UPGRADE\_NOWAIT or UPGRADE\_SKIPLOCKED, the 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. 10.4. Jakarta Persistence locking query hints Jakarta Persistence defined two locking-related query hints: jakarta.persistence.lock.timeout it gives the number of milliseconds a lock acquisition request will wait before throwing an exception jakarta.persistence.lock.scope defines the scope of the lock acquisition request. The scope can either be NORMAL (default value) or EXTENDED. The EXTENDED scope will cause a lock acquisition request to be passed to other owned table structured (e.g. @Inheritance(strategy=InheritanceType.JOINED), @ElementCollection) Example 445. jakarta.persistence.lock.timeout example entityManager.find(
Person.class, id, LockModeType.PESSIMISTIC\_WRITE,
Collections.singletonMap("jakarta.persistence.lock.timeout", 200)
); SELECT explicitlo0\_.id AS id1\_0\_0\_,
explicitlo0\_."name" AS name2\_0\_0\_
FROM person explicitlo0\_
WHERE explicitlo0\_.id = 1
FOR UPDATE wait 2 Not all JDBC database drivers support setting a timeout value for a locking request. If not supported, the Hibernate dialect ignores this query hint. The jakarta.persistence.lock.scope is not yet supported as specified by the Jakarta Persistence standard. 10.5. Session.lock() The following example shows how to obtain a shared database lock. Example 446. session.lock() example Person person = entityManager.find(Person.class, id);
Session session = entityManager.unwrap(Session.class);
LockOptions lockOptions = new LockOptions(LockMode.PESSIMISTIC\_READ, LockOptions.NO\_WAIT);
session.lock(person, lockOptions); SELECT p1\_0.id,
p1\_0."name"
FROM Person p1\_0
WHERE p1\_0.id = 1
SELECT id
FROM Person
WHERE id = 1
FOR UPDATE 10.6. Follow-on-locking When using Oracle, the FOR UPDATE exclusive locking clause cannot be used with: DISTINCT GROUP BY UNION inlined views (derived tables), therefore, affecting the legacy Oracle pagination mechanism as well. For this reason, Hibernate uses secondary selects to lock the previously fetched entities. Example 447. Follow-on-locking example List<Person> persons = entityManager.createQuery(
"select DISTINCT p from Person p", Person.class)
.setLockMode(LockModeType.PESSIMISTIC\_WRITE)
.getResultList(); SELECT DISTINCT p.id as id1\_0\_, p."name" as name2\_0\_
FROM Person p
SELECT id
FROM Person
WHERE id = 1 FOR UPDATE
SELECT id
FROM Person
WHERE id = 1 FOR UPDATE To avoid the N+1 query problem, a separate query can be used to apply the lock using the associated entity identifiers. Example 448. Secondary query entity locking List<Person> persons = entityManager.createQuery(
"select DISTINCT p from Person p", Person.class)
.getResultList();
entityManager.createQuery(
"select p.id from Person p where p in :persons")
.setLockMode(LockModeType.PESSIMISTIC\_WRITE)
.setParameter("persons", persons)
.getResultList(); SELECT DISTINCT p.id as id1\_0\_, p."name" as name2\_0\_
FROM Person p
SELECT p.id as col\_0\_0\_
FROM Person p
WHERE p.id IN ( 1 , 2 )
FOR UPDATE The lock request was moved from the original query to a secondary one which takes the previously fetched entities to lock their associated database records. Prior to Hibernate 5.2.1, the follow-on-locking mechanism was applied uniformly to any locking query executing on Oracle. Since 5.2.1, the Oracle Dialect tries to figure out if the current query demands the follow-on-locking mechanism. Even more important is that you can overrule the default follow-on-locking detection logic and explicitly enable or disable it on a per query basis. Example 449. Disabling the follow-on-locking mechanism explicitly List<Person> persons = entityManager.createQuery(
"select p from Person p", Person.class)
.setMaxResults(10)
.unwrap(Query.class)
.setLockOptions(
new LockOptions(LockMode.PESSIMISTIC\_WRITE)
.setFollowOnLocking(false))
.getResultList(); SELECT \*
FROM (
SELECT p.id as id1\_0\_, p."name" as name2\_0\_
FROM Person p
)
WHERE rownum <= 10
FOR UPDATE The follow-on-locking mechanism should be explicitly enabled only if the currently executing query fails because the FOR UPDATE clause cannot be applied, meaning that the Dialect resolving mechanism needs to be further improved.