6. FlushingFlushing is the process of synchronizing the state of the persistence context with the underlying database. The EntityManager and the Hibernate Session expose a set of methods, through which the application developer can change the persistent state of an entity. The persistence context acts as a transactional write-behind cache, queuing any entity state change. Like any write-behind cache, changes are first applied in-memory and synchronized with the database during the flush time. The flush operation takes every entity state change and translates it to an INSERT, UPDATE or DELETE statement. Because DML statements are grouped together, Hibernate can apply batching transparently. See the Batching chapter for more information. The flushing strategy is given by the flushMode of the current running Hibernate Session. Although Jakarta Persistence defines only two flushing strategies (AUTO and COMMIT), Hibernate has a much broader spectrum of flush types: ALWAYS Flushes the Session before every query. AUTO This is the default mode, and it flushes the Session only if necessary. COMMIT The Session tries to delay the flush until the current Transaction is committed, although it might flush prematurely too. MANUAL The Session flushing is delegated to the application, which must call Session.flush() explicitly in order to apply the persistence context changes. 6.1. AUTO flush By default, Hibernate uses the AUTO flush mode which triggers a flush in the following circumstances: prior to committing a Transaction prior to executing a JPQL/HQL query that overlaps with the queued entity actions before executing any native SQL query that has no registered synchronization 6.1.1. AUTO flush on commit In the following example, an entity is persisted, and then the transaction is committed. Example 419. Automatic flushing on commit entityManager = entityManagerFactory().createEntityManager();
txn = entityManager.getTransaction();
txn.begin();
Person person = new Person("John Doe");
entityManager.persist(person);
log.info("Entity is in persisted state");
txn.commit(); --INFO: Entity is in persisted state
INSERT INTO Person (name, id) VALUES ('John Doe', 1) Hibernate logs the message prior to inserting the entity because the flush only occurred during transaction commit. This is valid for the SEQUENCE and TABLE identifier generators. The IDENTITY generator must execute the insert right after calling persist(). For more details, see the discussion of generators in Identifier generators. 6.1.2. AUTO flush on JPQL/HQL query A flush may also be triggered when executing an entity query. Example 420. Automatic flushing on JPQL/HQL Person person = new Person("John Doe");
entityManager.persist(person);
entityManager.createQuery("select p from Advertisement p").getResultList();
entityManager.createQuery("select p from Person p").getResultList(); SELECT a.id AS id1\_0\_ ,
a.title AS title2\_0\_
FROM Advertisement a
INSERT INTO Person (name, id) VALUES ('John Doe', 1)
SELECT p.id AS id1\_1\_ ,
p.name AS name2\_1\_
FROM Person p The reason why the Advertisement entity query didn’t trigger a flush is that there’s no overlapping between the Advertisement and the Person tables: Example 421. Automatic flushing on JPQL/HQL entities @Entity(name = "Person")
public static class Person {
@Id
@GeneratedValue
private Long id;
private String name;
//Getters and setters are omitted for brevity
}
@Entity(name = "Advertisement")
public static class Advertisement {
@Id
@GeneratedValue
private Long id;
private String title;
//Getters and setters are omitted for brevity
} When querying for a Person entity, the flush is triggered prior to executing the entity query. Example 422. Automatic flushing on JPQL/HQL Person person = new Person("John Doe");
entityManager.persist(person);
entityManager.createQuery("select p from Person p").getResultList(); INSERT INTO Person (name, id) VALUES ('John Doe', 1)
SELECT p.id AS id1\_1\_ ,
p.name AS name2\_1\_
FROM Person p This time, the flush was triggered by a JPQL query because the pending entity persisting action overlaps with the query being executed. 6.1.3. AUTO flush on native SQL query When executing a native SQL query, a flush is always triggered when using the EntityManager API. Example 423. Automatic flushing on native SQL using EntityManager assertTrue(((Number) entityManager
.createNativeQuery("select count(\*) from Person")
.getSingleResult()).intValue() == 0);
Person person = new Person("John Doe");
entityManager.persist(person);
assertTrue(((Number) entityManager
.createNativeQuery("select count(\*) from Person")
.getSingleResult()).intValue() == 1); If you bootstrap Hibernate natively, and not through Jakarta Persistence, by default, the Session API will trigger a flush automatically when executing a native query. Example 424. Automatic flushing on native SQL using Session assertTrue(((Number) session
.createNativeQuery("select count(\*) from Person", Integer.class)
.getSingleResult()).intValue() == 0);
Person person = new Person("John Doe");
session.persist(person);
assertTrue(((Number) session
.createNativeQuery("select count(\*) from Person", Integer.class)
.uniqueResult()).intValue() == 0); To flush the Session, the query must use a synchronization: Example 425. Automatic flushing on native SQL with Session synchronization assertTrue(((Number) entityManager
.createNativeQuery("select count(\*) from Person")
.getSingleResult()).intValue() == 0);
Person person = new Person("John Doe");
entityManager.persist(person);
Session session = entityManager.unwrap(Session.class);
assertTrue(((Number) session
.createNativeQuery("select count(\*) from Person", Integer.class)
.addSynchronizedEntityClass(Person.class)
.uniqueResult()).intValue() == 1); 6.2. COMMIT flush Jakarta Persistence also defines a COMMIT flush mode, which is described as follows: If FlushModeType.COMMIT is set, the effect of updates made to entities in the persistence context upon queries is unspecified. — Section 3.10.8 of the Java Persistence 2.1 Specification When executing a JPQL query, the persistence context is only flushed when the current running transaction is committed. Example 426. COMMIT flushing on JPQL Person person = new Person("John Doe");
entityManager.persist(person);
entityManager.createQuery("select p from Advertisement p")
.setFlushMode(FlushModeType.COMMIT)
.getResultList();
entityManager.createQuery("select p from Person p")
.setFlushMode(FlushModeType.COMMIT)
.getResultList(); SELECT a.id AS id1\_0\_ ,
a.title AS title2\_0\_
FROM Advertisement a
SELECT p.id AS id1\_1\_ ,
p.name AS name2\_1\_
FROM Person p
INSERT INTO Person (name, id) VALUES ('John Doe', 1) Because the Jakarta Persistence doesn’t impose a strict rule on delaying flushing, when executing a native SQL query, the persistence context is going to be flushed. Example 427. COMMIT flushing on native SQL Person person = new Person("John Doe");
entityManager.persist(person);
assertTrue(((Number) entityManager
.createNativeQuery("select count(\*) from Person")
.getSingleResult()).intValue() == 1); INSERT INTO Person (name, id) VALUES ('John Doe', 1)
SELECT COUNT(\*) FROM Person 6.3. ALWAYS flush The ALWAYS is only available with the native Session API. The ALWAYS flush mode triggers a persistence context flush even when executing a native SQL query against the Session API. Example 428. COMMIT flushing on native SQL Person person = new Person("John Doe");
entityManager.persist(person);
Session session = entityManager.unwrap(Session.class);
assertTrue(((Number) session
.createNativeQuery("select count(\*) from Person", Integer.class)
.setHibernateFlushMode(FlushMode.ALWAYS)
.uniqueResult()).intValue() == 1); INSERT INTO Person (name, id) VALUES ('John Doe', 1)
SELECT COUNT(\*) FROM Person 6.4. MANUAL flush Both the EntityManager and the Hibernate Session define a flush() method that, when called, triggers a manual flush. Hibernate also provides a MANUAL flush mode so the persistence context can only be flushed manually. Example 429. MANUAL flushing Person person = new Person("John Doe");
entityManager.persist(person);
Session session = entityManager.unwrap(Session.class);
session.setHibernateFlushMode(FlushMode.MANUAL);
assertTrue(((Number) entityManager
.createQuery("select count(id) from Person")
.getSingleResult()).intValue() == 0);
assertTrue(((Number) session
.createNativeQuery("select count(\*) from Person", Integer.class)
.uniqueResult()).intValue() == 0); SELECT COUNT(p.id) AS col\_0\_0\_
FROM Person p
SELECT COUNT(\*)
FROM Person The INSERT statement was not executed because there was no manual flush() call. The MANUAL flush mode is useful when using multi-request logical transactions, and only the last request should flush the persistence context. 6.5. Flush operation order From a database perspective, a row state can be altered using either an INSERT, an UPDATE or a DELETE statement. Because entity state changes are automatically converted to SQL statements, it’s important to know which entity actions are associated with a given SQL statement. INSERT The INSERT statement is generated either by the EntityInsertAction or EntityIdentityInsertAction. These actions are scheduled by the persist operation, either explicitly or through cascading the PersistEvent from a parent to a child entity. DELETE The DELETE statement is generated by the EntityDeleteAction or OrphanRemovalAction. UPDATE The UPDATE statement is generated by EntityUpdateAction during flushing if the managed entity has been marked modified. The dirty checking mechanism is responsible for determining if a managed entity has been modified since it was first loaded. Hibernate does not execute the SQL statements in the order of their associated entity state operations. To visualize how this works, consider the following example: Example 430. Flush operation order Person person = entityManager.find(Person.class, 1L);
entityManager.remove(person);
Person newPerson = new Person();
newPerson.setId(2L);
newPerson.setName("John Doe");
entityManager.persist(newPerson); INSERT INTO Person (name, id)
VALUES ('John Doe', 2L)
DELETE FROM Person WHERE id = 1 Even if we removed the first entity and then persist a new one, Hibernate is going to execute the DELETE statement after the INSERT. The order in which SQL statements are executed is given by the ActionQueue and not by the order in which entity state operations have been previously defined. The ActionQueue executes all operations in the following order: OrphanRemovalAction EntityInsertAction or EntityIdentityInsertAction EntityUpdateAction QueuedOperationCollectionAction CollectionRemoveAction CollectionUpdateAction CollectionRecreateAction EntityDeleteAction