The Persistence Context and Locking

Why dirty checking, already covered in full in "Transaction Management," actually works -- the persistence context itself, entity lifecycle states (transient/managed/detached /removed), the first-level cache, persist/merge/detach, flush timing -- and what happens when two transactions touch the same row: optimistic locking with @Version, pessimistic locking with @Lock. The 7th lesson in the Spring Data JPA category.

Advanced 35 min

Last updated: August 24, 2026
Read online: https://www.learnforgex.com/en/topics/persistence-context-and-locking

---

"Transaction Management" already covered `@Transactional` in full — propagation, isolation, rollback rules, and briefly, that Hibernate's dirty checking writes a changed field to the database at commit time even without an explicit `save()` call. It left one question open: WHY does that work at all? This lesson answers it — the persistence context itself, the object that makes dirty checking (and a handful of other things) possible — and then covers what happens when two transactions genuinely collide over the same row.

What "Transaction Management" Already Covered, and What This Lesson Adds

`@Transactional`'s propagation, isolation levels, rollback rules, and the proxy mechanism behind it are already covered in full — none of that is repeated here. Dirty checking itself was already named and demonstrated conceptually there too. What's new: the persistence context that dirty checking actually runs inside of, the full lifecycle an entity moves through within it, and — genuinely new territory — what happens when two separate transactions modify the same row at the same time.

The Persistence Context: More Than Just "the Current Transaction"

The persistence context is the object (backed by JPA's `EntityManager`) that tracks every entity a transaction has loaded or saved — it's what "Transaction Management" was really describing when it said a managed entity's field changes get written back automatically. This project's own code never touches `EntityManager` directly — every repository method (`save`, `findById`, `delete`) already wraps it — but `repository.save(...)` is itself built on the exact operations this lesson covers directly.

Entity States: Transient, Managed, Detached, Removed

Every entity instance is, at any given moment, in exactly one of four states relative to the persistence context.

```
import jakarta.persistence.Entity;
import jakarta.persistence.EntityManager;
import jakarta.persistence.Id;

// This project's own code never touches EntityManager directly -- every
// repository method (save, findById, delete) already wraps it. But
// repository.save(...) is ITSELF built on the operations shown here --
// understanding them is what makes dirty checking (already covered in
// "Transaction Management") make sense as something more than "magic."
@Entity
class TopicLifecycleExample {
    @Id
    private Long id;
    private String slug;

    TopicLifecycleExample() {
    }

    TopicLifecycleExample(Long id, String slug) {
        this.id = id;
        this.slug = slug;
    }
}

class EntityLifecycleExample {

    static void demonstrateLifecycle(EntityManager em) {
        // TRANSIENT: a plain Java object, "new"'d up -- the persistence
        // context has never heard of it, and nothing about it is tracked.
        TopicLifecycleExample topic = new TopicLifecycleExample(1L, "records");

        // MANAGED: persist(...) hands this object to the persistence
        // context -- from this point on, any field change is tracked, and
        // will be written to the database at the next flush, with no
        // explicit save() call required (this IS dirty checking).
        em.persist(topic);
        topic.slug = "record"; // tracked automatically -- no save() needed

        // DETACHED: once the persistence context closes (the transaction
        // ends) or detach(...) is called explicitly, the entity is no
        // longer tracked -- further field changes are NOT written back
        // automatically anymore.
        em.detach(topic);
        topic.slug = "ignored-change"; // NOT tracked -- topic is detached now

        // REMOVED: remove(...) schedules a managed entity for deletion at
        // the next flush -- it must be re-attached (merged) first if it's
        // currently detached, since remove(...) only works on managed entities.
        TopicLifecycleExample managedAgain = em.merge(topic);
        em.remove(managedAgain);
    }
}
```

TRANSIENT is a plain Java object the persistence context has never heard of — `new TopicLifecycleExample(...)` alone. MANAGED means the persistence context is actively tracking it — `em.persist(...)` makes that transition, and from that point on, field changes

are tracked automatically (this IS dirty checking, now with a name for the mechanism behind it). DETACHED means the persistence context no longer tracks it — either the transaction ended, or `detach(...)` was called explicitly — and further field changes are NOT written back automatically anymore. REMOVED means `remove(...)` has scheduled a managed entity for deletion at the next flush; a detached entity has to be re-attached with `merge(...)` first, since `remove(...)` only accepts managed entities.

The First-Level Cache: Why the Same Query Can Return the Same Object

Within a single persistence context, asking for the same entity twice doesn't necessarily mean two trips to the database.

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import org.springframework.data.jpa.repository.JpaRepository;

@Entity
class TopicCacheExample {
    @Id
    private Long id;
    private String slug;
}

interface TopicCacheRepositoryExample extends JpaRepository<TopicCacheExample, Long> {
}

class FirstLevelCacheExample {

    static void demonstrateIdentityMap(TopicCacheRepositoryExample repository) {
        // Within the SAME persistence context (the same transaction), two
        // separate calls asking for the SAME id don't run two separate
        // SELECT queries returning two separate Java objects -- the
        // persistence context's first-level cache recognizes "id 5" is
        // already being tracked, and hands back the EXACT SAME instance.
        TopicCacheExample first = repository.findById(5L).orElseThrow();
        TopicCacheExample second = repository.findById(5L).orElseThrow();

        System.out.println(first == second);
        // true -- not just equal, but the literal same object reference.
        // Only the FIRST call actually queried the database; the second
        // was answered entirely from the first-level cache.
    }
}
```

Two separate `repository.findById(5L)` calls, within the same transaction, don't run two separate queries returning two separate objects — the persistence context's first-level cache recognizes that entity `5` is already being tracked and hands back the EXACT SAME instance. Only the first call actually queries the database; the second is answered entirely

from memory. This cache is scoped to a single persistence context — it doesn't span transactions, and it isn't the kind of shared, cross-request cache a "second-level cache" would be.

persist(), merge(), and detach()

`persist()` isn't the only way to bring an entity under the persistence context's management — a detached entity, one that already has an id but isn't currently tracked, needs a different operation entirely.

```
import jakarta.persistence.Entity;
import jakarta.persistence.EntityManager;
import jakarta.persistence.Id;

@Entity
class TopicMergeExample {
    @Id
    private Long id;
    private String slug;

    TopicMergeExample() {
    }

    TopicMergeExample(Long id, String slug) {
        this.id = id;
        this.slug = slug;
    }
}

class PersistMergeDetachExample {

    // A realistic scenario: a Topic loaded in one request (or one layer),
    // then modified and handed to a DIFFERENT persistence context later
    // -- for instance, a DTO-to-entity conversion happening outside the
    // original transaction. This "topic" object is DETACHED: it has an
    // id, but this EntityManager has never seen it.
    static void updateDetachedTopic(EntityManager em, TopicMergeExample detachedTopic) {
        // persist(...) would be WRONG here -- it's for entities that have
        // never existed in the database at all; calling it on an object
        // that already has an id representing an existing row risks a
        // duplicate-key error rather than an update.

        // merge(...) is the correct operation for a detached entity: it
        // copies this object's field values onto a MANAGED entity (loading
        // it first if needed) and returns THAT managed entity -- the
        // original "detachedTopic" parameter itself stays detached and
        // untracked.
        TopicMergeExample managed = em.merge(detachedTopic);

        managed.slug = "updated-via-merge"; // tracked, because "managed" is managed
        // detachedTopic.slug = "ignored";   // would NOT be tracked -- still detached
    }
}
```

`persist(...)` is for entities that have never existed in the database — calling it on an object that already represents an existing row risks a duplicate-key error rather than an update. `merge(...)` is the correct operation for a detached entity: it copies the object's field values onto a managed entity (loading it first if needed) and returns THAT managed entity — the original detached object handed to `merge(...)` stays detached and untracked; only the returned object is actually managed.

Flushing: When Changes Actually Reach the Database

A flush is the specific moment tracked changes are sent to the database as real SQL — a distinct moment from both the change itself and the transaction's eventual commit.

```
import jakarta.persistence.Entity;
import jakarta.persistence.EntityManager;
import jakarta.persistence.Id;
import jakarta.persistence.Query;

@Entity
class TopicFlushExample {
    @Id
    private Long id;
    private String difficulty;
}

class FlushTimingExample {

    // A flush is when tracked changes are actually SENT to the database
    // as SQL -- not the same moment as the change itself (setting a
    // field) or the same moment as the transaction committing.
    static void demonstrateAutoFlush(EntityManager em) {
        TopicFlushExample topic = em.find(TopicFlushExample.class, 1L);
        topic.difficulty = "ADVANCED"; // tracked, not yet sent to the database

        // Hibernate AUTO-FLUSHES before running a query whose result could
        // be affected by pending changes -- this JPQL query touches the
        // "difficulty" column, so Hibernate flushes the change above FIRST,
        // to guarantee the query sees it, even though flush() was never
        // called explicitly.
        Query query = em.createQuery("select t from TopicFlushExample t where t.difficulty
= 'ADVANCED'");
        query.getResultList(); // topic (id=1) is included -- the flush already happened

        // An EXPLICIT flush forces this same thing to happen right now,
        // without waiting for a query or the transaction's own commit --
        // useful when code genuinely needs to know a change has reached
        // the database before continuing, without ending the transaction.
        em.flush();
    }
}
```

Hibernate auto-flushes before running a query whose result could be affected by pending changes — a JPQL query filtering on `difficulty` triggers a flush of a pending `difficulty`

change first, automatically, so the query's own result reflects it, even though `flush()` was never called explicitly. An explicit `em.flush()` forces that same thing to happen immediately, without waiting for a query to trigger it or the transaction to commit — useful when code genuinely needs a change to have reached the database before continuing, without ending the transaction itself.

Optimistic Locking with @Version

Dirty checking assumes a single transaction is the only one touching a row. Real applications can't always assume that — two transactions can load, and both try to modify, the exact same row.

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Version;
import org.springframework.dao.OptimisticLockingFailureException;
import org.springframework.data.jpa.repository.JpaRepository;

@Entity
class TopicVersionExample {
    @Id
    private Long id;
    private Integer estimatedMinutes;

    // @Version adds a column Hibernate manages entirely on its own -- every
    // UPDATE increments it, and every UPDATE's WHERE clause checks it still
    // matches the value this entity was loaded with:
    //   UPDATE topic SET estimated_minutes = ?, version = version + 1
    //   WHERE id = ? AND version = ?  (the version read at load time)
    @Version
    private Integer version;

    void setEstimatedMinutes(int minutes) {
        this.estimatedMinutes = minutes;
    }
}

interface TopicVersionRepositoryExample extends JpaRepository<TopicVersionExample, Long> {
}

class OptimisticLockingExample {

    // "Optimistic" because it assumes collisions are RARE -- no lock is
    // held while the entity is read and modified; the check only happens
    // at write time, at essentially no cost when nothing actually collides.
    static void demonstrateCollision(TopicVersionRepositoryExample repository) {
        TopicVersionExample loadedByUserA = repository.findById(1L).orElseThrow(); //
version = 3
        TopicVersionExample loadedByUserB = repository.findById(1L).orElseThrow(); // ALSO
version = 3

        loadedByUserA.setEstimatedMinutes(30);
        repository.save(loadedByUserA); // succeeds -- version 3 still matches; row becomes
version 4

        loadedByUserB.setEstimatedMinutes(45);
        try {
            repository.save(loadedByUserB); // the WHERE ... AND version = 3 now matches NO
row
        } catch (OptimisticLockingFailureException e) {
            // User B's save fails -- the row's version is 4 now, not 3, so
            // the UPDATE's WHERE clause matched zero rows, and Spring Data
            // JPA reports that as this exception instead of silently doing
            // nothing.
            System.out.println("Someone else already changed this topic: " + e.
getMessage());
        }
    }
}
```

`@Version` adds a column Hibernate manages entirely on its own — every `UPDATE` increments it, and every `UPDATE`'s `WHERE` clause checks it still matches the value the entity was loaded with. It's called OPTIMISTIC because it assumes collisions are rare — no lock is held while an entity is read and modified; the check only happens at write time, at essentially no cost when nothing actually collides.

What Happens When Two Transactions Collide

When a collision does happen, the second write's `WHERE ... AND version = ...` clause matches zero rows — the row's version has already moved on.

`loadedByUserA` and `loadedByUserB` both load the same row at `version = 3`. User A saves first — the `UPDATE`'s `WHERE version = 3` still matches, so it succeeds, and the row becomes `version = 4`. User B saves next, still believing the version is `3` — the `UPDATE`'s `WHERE version = 3` now matches nothing at all, and Spring Data JPA surfaces that as an `OptimisticLockingFailureException` rather than silently doing nothing or overwriting User A's change.

> An `OptimisticLockingFailureException` isn't a bug to suppress — it's the system correctly detecting that the data a user was working from is now stale. The typical response is showing the user the current data and asking them to reapply their change, not blindly retrying the same save.

Pessimistic Locking with @Lock

Optimistic locking detects a collision after it happens. A different strategy prevents the collision from being possible in the first place.

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.LockModeType;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Lock;
import org.springframework.data.jpa.repository.Query;

@Entity
class TopicLockExample {
    @Id
    private Long id;
    private Integer estimatedMinutes;
}

interface TopicLockRepositoryExample extends JpaRepository<TopicLockExample, Long> {

    // "Pessimistic" because it assumes a collision is LIKELY enough to
    // prevent outright, rather than detect after the fact -- @Lock adds a
    // real database-level lock (PostgreSQL's SELECT ... FOR UPDATE) at
    // READ time. Any other transaction trying to read this same row with
    // its own PESSIMISTIC_WRITE lock simply WAITS until this transaction
    // commits or rolls back.
    @Lock(LockModeType.PESSIMISTIC_WRITE)
    @Query("select t from TopicLockExample t where t.id = :id")
    TopicLockExample findByIdForUpdate(Long id);
}

// Optimistic locking (the default, everyday choice) detects a collision
// after it happens and rejects the losing write. Pessimistic locking
// prevents the collision from being possible in the first place, at the
// cost of making every other transaction wait -- reach for it only for
// genuinely high-contention operations (a shared counter, a seat
// reservation), not as a default.
```

`@Lock(LockModeType.PESSIMISTIC_WRITE)` adds a real database-level lock (PostgreSQL's `SELECT ... FOR UPDATE`) at read time — any other transaction trying to acquire the same lock on the same row simply WAITS until this transaction commits or rolls back. It's called PESSIMISTIC because it assumes a collision is likely enough to prevent outright, at the cost of making every other transaction wait — reach for it for genuinely high-contention operations (a shared counter, a seat reservation), not as a default in place of optimistic locking.

Common Misconceptions

"Dirty checking just happens, with no real mechanism behind it." It's the persistence context doing exactly what "managed" means — tracking a specific entity's field changes for exactly as long as it stays managed. **"The first-level cache is the same as a general application-wide cache."** It isn't — it's scoped to one persistence context (one transaction, in the common case), never shared across requests or transactions.

"Optimistic and pessimistic locking are two names for the same idea." They're opposite strategies — one detects a collision after it happens and rejects the loser; the other prevents the collision from being possible at all, by making everyone else wait.

What Comes Next

Every entity covered in this category so far has been a plain Java field — a `String`, an `Integer`, an enum, a relationship. "Auditing in Spring Data JPA," next in this category, covers a specific, common kind of field this lesson's entity-state mechanics make possible: automatically recording WHEN an entity was created or last modified, and by WHOM, without writing that logic by hand in every service method.

Best Practices

- Reach for `merge()` specifically when working with a detached entity — using `persist()` on one risks a duplicate-key error instead of the update you actually wanted.
- Add `@Version` to any entity genuinely at risk of concurrent modification by more than one transaction — it costs almost nothing when collisions don't happen, and catches real ones when they do.
- Handle `OptimisticLockingFailureException` by re-showing the current data to whoever's making the change, not by silently retrying the same save.
- Reserve pessimistic locking for genuinely high-contention operations — its cost (making other transactions wait) isn't worth paying by default.

Common Mistakes

- Calling `persist()` on a detached entity instead of `merge()`, risking a duplicate-key error rather than the intended update.
- Assuming the first-level cache behaves like a general, cross-request cache, rather than something scoped to a single persistence context.
- Catching `OptimisticLockingFailureException` and silently retrying the exact same save — this just repeats the same stale-data problem.
- Reaching for `@Lock(PESSIMISTIC_WRITE)` by default, making every transaction touching that row wait, when `@Version`-based optimistic locking would have been enough.

Summary, Cheat Sheet, and Glossary

Summary

- The persistence context tracks every entity a transaction has loaded or saved — it's what makes dirty checking, already covered in "Transaction Management," possible in the first place.

- An entity is transient (untracked, new), managed (tracked), detached (no longer tracked), or removed (scheduled for deletion) — never more than one at a time.
- The first-level cache means asking for the same entity twice, within one persistence context, can return the exact same object without a second query.
- `persist()` is for entities that never existed in the database; `merge()` is for detached entities that already do.
- A flush sends tracked changes to the database as SQL — automatically before a query that needs to see them, or explicitly via `flush()`.
- `@Version` (optimistic locking) detects a collision between two transactions after it happens; `@Lock(PESSIMISTIC_WRITE)` prevents the collision from being possible at all.

Cheat Sheet

```
// Entity lifecycle
Topic topic = new Topic();        // transient
em.persist(topic);                // managed
em.detach(topic);                 // detached
Topic managed = em.merge(topic);  // managed again
em.remove(managed);               // removed

// First-level cache: same id, same object, within one persistence context
repository.findById(5L) == repository.findById(5L) // true

// Optimistic locking
@Version
private Integer version;
// UPDATE ... WHERE id = ? AND version = ?  (fails if the row already moved on)

// Pessimistic locking
@Lock(LockModeType.PESSIMISTIC_WRITE)
@Query("select t from Topic t where t.id = :id")
Topic findByIdForUpdate(Long id);
```

Glossary

- **Persistence context**: the object tracking every entity a transaction has loaded or saved, enabling dirty checking and the first-level cache.
- **Entity state**: transient, managed, detached, or removed — an entity's current relationship to the persistence context.
- **First-level cache**: the persistence context's own identity map, returning the same object instance for the same id within one persistence context.
- **Flush**: the moment tracked changes are sent to the database as SQL, distinct from both the change itself and the transaction's commit.
- **Optimistic locking (@Version)**: detecting a collision between two transactions after it happens, via a version column checked on every update.
- **Pessimistic locking (@Lock)**: preventing a collision by holding a real database lock, making other transactions wait instead of detecting a conflict afterward.