The Persistence Context and Locking

"Transaction Management"in zaten kapsamlıca işlediği dirty checking'in NEDEN çalıştığı -- persistence context'in kendisi, entity yaşam döngüsü durumları (transient/managed /detached/removed), first-level cache, persist/merge/detach, flush zamanlaması -- ve iki transaction aynı satıra dokunduğunda ne olduğu: @Version ile optimistic locking, @Lock ile pessimistic locking. Spring Data JPA kategorisinin 7.'si.

İleri 35 dk

Son güncelleme: 24 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/persistence-context-and-locking

---

"Transaction Management", `@Transactional`'ı -- propagation, isolation, rollback kuralları, ve kısaca, Hibernate'in dirty checking'inin, açık bir `save()` çağrısı olmadan bile, değişen bir alanı commit zamanında veritabanına yazdığını -- zaten tam olarak işledi. Bir soruyu açık bıraktı: bu NEDEN çalışıyor? Bu ders bunu cevaplıyor -- dirty checking'i (ve bir avuç başka şeyi) mümkün kılan şeyin kendisi, persistence context -- ve sonra iki transaction gerçekten aynı satır üzerinde çarpıştığında ne olduğunu işliyor.

"Transaction Management"in Zaten İşlediği, ve Bu Dersin Eklediği

`@Transactional`'ın propagation'ı, isolation seviyeleri, rollback kuralları, ve arkasındaki proxy mekanizması zaten tam olarak işlendi -- hiçbiri burada tekrarlanmıyor. Dirty checking'in kendisi de orada zaten kavramsal olarak adlandırılıp gösterildi. Yeni olan: dirty checking'in gerçekte içinde çalıştığı persistence context, bir entity'nin onun içinde geçtiği tam yaşam döngüsü, ve -- gerçekten yeni bir alan -- iki ayrı transaction aynı satırı aynı anda değiştirdiğinde ne olduğu.

Persistence Context: Yalnızca "Mevcut Transaction"dan Fazlası

Persistence context, bir transaction'ın yüklediği ya da kaydettiği her entity'yi izleyen (JPA'nın `EntityManager`'ı tarafından desteklenen) nesnedir -- "Transaction Management"in, managed bir entity'nin alan değişikliklerinin otomatik olarak geri yazıldığını söylerken gerçekte tarif ettiği şey buydu. Bu projenin kendi kodu `EntityManager`'a asla doğrudan dokunmaz -- her repository metodu (`save`, `findById`, `delete`) onu zaten sarmalıyor -- ama `repository.save(...)`'ın kendisi, bu dersin doğrudan işlediği tam olarak aynı işlemlerin üzerine inşa edilmiştir.

Entity Durumları: Transient, Managed, Detached, Removed

Her entity instance'ı, herhangi bir anda, persistence context'e göre tam olarak dört durumdan birindedir.

```
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, persistence context'in hiç haberdar olmadığı düz bir Java nesnesidir -- yalnızca `new TopicLifecycleExample(...)`. MANAGED, persistence context'in onu aktif olarak izlediği anlamına gelir -- `em.persist(...)` bu geçişi yapar, ve o andan itibaren, alan değişiklikleri

otomatik olarak izlenir (bu, artık arkasındaki mekanizmaya bir isim verilmiş dirty checking'in KENDİSİDİR). DETACHED, persistence context'in onu artık izlemediği anlamına gelir -- ya transaction sona erdi, ya da `detach(...)` açıkça çağrıldı -- ve sonraki alan değişiklikleri artık otomatik olarak geri yazılmaz. REMOVED, `remove(...)`'un managed bir entity'yi bir sonraki flush'ta silinmek üzere planladığı anlamına gelir; detached bir entity'nin önce `merge(...)` ile yeniden bağlanması gerekir, çünkü `remove(...)` yalnızca managed entity'leri kabul eder.

First-Level Cache: Aynı Sorgu Neden Aynı Nesneyi Döndürebilir

Tek bir persistence context içinde, aynı entity'yi iki kez istemek, veritabanına iki gidiş anlamına gelmek zorunda değildir.

```
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.
    }
}
```

Aynı transaction içindeki iki ayrı `repository.findById(5L)` çağrısı, iki ayrı nesne döndüren iki ayrı sorgu çalıştırmaz -- persistence context'in first-level cache'i, `5` entity'sinin zaten izlendiğini tanır ve TAM OLARAK AYNI instance'ı geri verir. Yalnızca ilk çağrı gerçekten veritabanını sorgular; ikincisi tamamen bellekten cevaplanır. Bu cache, tek bir persistence

context'e (genel durumda, tek bir transaction'a) kapsam sınırlıdır -- transaction'lar arasında yayılmaz, ve bir "second-level cache"in olacağı türden paylaşılan, istekler-arası bir cache değildir.

persist(), merge() ve detach()

`persist()`, bir entity'yi persistence context'in yönetimine getirmenin tek yolu değildir -- zaten bir id'si olan ama şu anda izlenmeyen, detached bir entity, tamamen farklı bir işleme ihtiyaç duyar.

```
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()`, veritabanında hiç var olmamış entity'ler içindir -- zaten var olan bir satırı temsil eden bir nesne üzerinde çağırmak, bir güncelleme yerine bir duplicate-key hatası riski taşır. `merge()`, detached bir entity için doğru işlemdir: nesnenin alan değerlerini managed bir entity'ye (gerekirse önce onu yükleyerek) kopyalar ve O managed entity'yi döndürür -- `merge(...)`'e verilen orijinal detached nesne detached ve izlenmeyen kalır; yalnızca döndürülen nesne gerçekten managed'dir.

Flush: Değişikliklerin Veritabanına Gerçekte Ulaştığı An

Bir flush, izlenen değişikliklerin gerçek SQL olarak veritabanına gönderildiği belirli andır -- hem değişikliğin kendisinden hem de transaction'ın nihai commit'inden ayrı bir an.

```
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, sonucu bekleyen değişikliklerden etkilenebilecek bir sorgu çalıştırmadan önce otomatik olarak flush eder -- `difficulty` üzerinde filtreleyen bir JPQL sorgusu, sorgunun

kendi sonucunun onu yansıtması için, bekleyen bir `difficulty` değişikliğinin flush'ını önce, otomatik olarak tetikler, `flush()` hiç açıkça çağrılmamış olsa bile. Açık bir `em.flush()`, bunu bir sorgunun tetiklemesini ya da transaction'ın commit olmasını beklemeden, hemen şimdi zorlar -- kod, transaction'ın kendisini sonlandırmadan, devam etmeden önce bir değişikliğin gerçekten veritabanına ulaştığını bilmesi gerektiğinde yararlıdır.

@Version ile Optimistic Locking

Dirty checking, bir satıra dokunan tek transaction'ın kendisi olduğunu varsayar. Gerçek uygulamalar her zaman bunu varsayamaz -- iki transaction aynı satırı yükleyip ikisi de değiştirmeye çalışabilir.

```
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`, Hibernate'in tamamen kendi başına yönettiği bir sütun ekler -- her `UPDATE` onu artırır, ve her `UPDATE`'in `WHERE` cümlesi, entity'nin yüklendiği değerle hâlâ eşleştiğini kontrol eder. OPTIMISTIC olarak adlandırılır çünkü çakışmaların nadir olduğunu varsayar -- bir entity okunup değiştirilirken hiçbir kilit tutulmaz; kontrol yalnızca yazma zamanında gerçekleşir, gerçekte hiçbir şey çakışmadığında neredeyse hiçbir maliyeti yoktur.

İki Transaction Çarpıştığında Ne Olur

Bir çarpışma gerçekten olduğunda, ikinci yazmanın `WHERE ... AND version = ...` cümlesi hiçbir satırla eşleşmez -- satırın version'ı zaten ilerlemiştir.

`loadedByUserA` ve `loadedByUserB`, ikisi de aynı satırı `version = 3`'te yükler. Kullanıcı A önce kaydeder -- `UPDATE`'in `WHERE version = 3`'ü hâlâ eşleşir, bu yüzden başarılı olur, ve satır `version = 4` olur. Kullanıcı B sonra kaydeder, hâlâ version'ın `3` olduğuna inanarak -- `UPDATE`'in `WHERE version = 3`'ü artık hiçbir şeyle eşleşmez, ve Spring Data JPA bunu sessizce hiçbir şey yapmamak ya da Kullanıcı A'nın değişikliğinin üzerine yazmak yerine bir `OptimisticLockingFailureException` olarak ortaya çıkarır.

> Bir `OptimisticLockingFailureException`, bastırılacak bir hata değildir -- sistemin, bir kullanıcının üzerinde çalıştığı verinin artık eski olduğunu doğru şekilde tespit etmesidir. Tipik tepki, aynı kaydetmeyi körü körüne yeniden denemek değil, kullanıcıya mevcut veriyi göstermek ve değişikliğini yeniden uygulamasını istemektir.

@Lock ile Pessimistic Locking

Optimistic locking, bir çarpışmayı gerçekleştikten sonra tespit eder. Farklı bir strateji, çarpışmanın baştan mümkün olmasını önler.

```
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)`, okuma zamanında gerçek bir veritabanı-seviyesi kilit (PostgreSQL'in `SELECT ... FOR UPDATE`'i) ekler -- aynı satırda aynı kilidi almaya çalışan başka herhangi bir transaction, bu transaction commit olana ya da rollback olana kadar basitçe BEKLER. PESSIMISTIC olarak adlandırılır çünkü bir çarpışmanın, her diğer transaction'ı bekletmek pahasına, baştan önlenmeye yetecek kadar olası olduğunu varsayar -- buna, optimistic locking'in varsayılan bir yerine geçeni olarak değil, gerçekten yüksek-çekişmeli işlemler (paylaşılan bir sayaç, bir koltuk rezervasyonu) için başvur.

Yaygın Yanlış Anlamalar

"Dirty checking arkasında gerçek bir mekanizma olmadan kendiliğinden olur." Persistence context'in tam olarak "managed" olmanın anlamını yaptığı şeydir -- belirli bir entity'nin alan değişikliklerini, managed kaldığı sürece tam olarak izlemek. **"First-level cache, genel bir uygulama-geneli cache ile aynıdır."** Değildir -- tek bir persistence context'e (yaygın durumda, tek bir transaction'a) kapsam sınırlıdır, istekler ya da

transaction'lar arasında asla paylaşılmaz. **"Optimistic ve pessimistic locking, aynı fikrin iki ismidir."** Zıt stratejilerdir -- biri bir çarpışmayı gerçekleştikten sonra tespit edip kaybedeni reddeder; diğeri, herkesi beklemeye zorlayarak çarpışmayı baştan imkansız kılar.

Sırada Ne Var

Bu kategoride şimdiye kadar işlenen her entity, düz bir Java alanıydı -- bir `String`, bir `Integer`, bir enum, bir ilişki. Bu kategoride sıradaki "Spring Data JPA'da Auditing", bu dersin entity-durumu mekaniğinin mümkün kıldığı, belirli, yaygın bir alan türünü işliyor: bir entity'nin NE ZAMAN oluşturulduğunu ya da son değiştirildiğini, ve KİM TARAFINDAN, bunu her servis metodunda elle yazmadan otomatik olarak kaydetmek.

Best Practices

- Detached bir entity'yle çalışırken özellikle `merge()`'e başvur -- onun üzerinde `persist()` kullanmak, gerçekten istediğin güncelleme yerine bir duplicate-key hatası riski taşır.
- Birden fazla transaction tarafından eş zamanlı değiştirilme riski gerçekten olan herhangi bir entity'ye `@Version` ekle -- çarpışma olmadığında neredeyse hiçbir maliyeti yoktur, ve gerçek olanları yakalar.
- `OptimisticLockingFailureException`'ı, aynı kaydetmeyi sessizce yeniden deneyerek değil, değişikliği yapan kişiye mevcut veriyi yeniden göstererek ele al.
- Pessimistic locking'i gerçekten yüksek-çekişmeli işlemler için sakla -- maliyeti (diğer transaction'ları bekletmek), varsayılan olarak ödenmeye değmez.

Yaygın Hatalar

- Detached bir entity üzerinde `merge()` yerine `persist()` çağırmak, istenen güncelleme yerine bir duplicate-key hatası riski taşımak.
- First-level cache'in, tek bir persistence context'e kapsam sınırlı bir şey yerine, genel, istekler-arası bir cache gibi davrandığını varsaymak.
- `OptimisticLockingFailureException`'ı yakalayıp aynı kaydetmeyi sessizce yeniden denemek -- bu yalnızca aynı eski-veri sorununu tekrarlar.
- `@Version`-tabanlı optimistic locking yeterli olacakken varsayılan olarak `@Lock(PESSIMISTIC_WRITE)`'a başvurmak, o satıra dokunan her transaction'ı bekletmek.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Persistence context, bir transaction'ın yüklediği ya da kaydettiği her entity'yi izler -- "Transaction Management"te zaten işlenen dirty checking'i baştan mümkün kılan şey budur.

- Bir entity, transient (izlenmeyen, yeni), managed (izlenen), detached (artık izlenmeyen), ya da removed (silinmek üzere planlanmış)dir -- asla aynı anda birden fazlası değil.
- First-level cache, tek bir persistence context içinde aynı entity'yi iki kez istemenin, ikinci bir sorgu olmadan tam olarak aynı nesneyi döndürebileceği anlamına gelir.
- `persist()`, veritabanında hiç var olmamış entity'ler içindir; `merge()`, zaten var olan detached entity'ler içindir.
- Bir flush, izlenen değişiklikleri SQL olarak veritabanına gönderir -- onları görmesi gereken bir sorgudan önce otomatik olarak, ya da `flush()` ile açıkça.
- `@Version` (optimistic locking), iki transaction arasındaki bir çarpışmayı gerçekleştikten sonra tespit eder; `@Lock(PESSIMISTIC_WRITE)`, çarpışmayı baştan imkansız kılar.

Cheat Sheet

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

// First-level cache: aynı id, aynı nesne, tek bir persistence context içinde
repository.findById(5L) == repository.findById(5L) // true

// Optimistic locking
@Version
private Integer version;
// UPDATE ... WHERE id = ? AND version = ?  (satır zaten ilerlediyse başarısız olur)

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

Terimler Sözlüğü

- **Persistence context**: bir transaction'ın yüklediği ya da kaydettiği her entity'yi izleyen, dirty checking'i ve first-level cache'i mümkün kılan nesne.
- **Entity durumu**: transient, managed, detached, ya da removed -- bir entity'nin persistence context'le mevcut ilişkisi.
- **First-level cache**: persistence context'in kendi identity map'i, tek bir persistence context içinde aynı id için aynı nesne instance'ını döndürür.
- **Flush**: izlenen değişikliklerin SQL olarak veritabanına gönderildiği an, hem değişikliğin kendisinden hem de transaction'ın commit'inden ayrı.
- **Optimistic locking (@Version)**: iki transaction arasındaki bir çarpışmayı, her güncellemede kontrol edilen bir version sütunuyla, gerçekleştikten sonra tespit etmek.
- **Pessimistic locking (@Lock)**: bir çarpışmayı, sonradan bir çakışmayı tespit etmek yerine diğer transaction'ları bekleterek, gerçek bir veritabanı kilidi tutarak önlemek.