SPRING DATA JPA

High-Performance & Advanced Techniques

CHAKIR CHAIMAE

Dynamic SQL Optimization

@DynamicUpdate

Generates SQL UPDATE statements with only modified fields, reducing database overhead. Without it, all fields are updated—even if unchanged.

@DynamicInsert

Generates SQL INSERT statements with only non-null fields, avoiding unnecessary NULLs and improving efficiency.

```java
@Entity
@DynamicUpdate // Only modified fields are included in UPDATE
@DynamicInsert // Only non-null fields are included in INSERT
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private BigDecimal price;
    private String description;
}

// Example usage
Product product = repo.findById(1L);
product.setPrice(new BigDecimal("19.99"));
repo.save(product);
/*
WITH @DynamicUpdate:
UPDATE product SET price = ? WHERE id = ?

WITHOUT @DynamicUpdate:
UPDATE product SET name = ?, price = ?, description = ?, id = ? WHERE id = ?
*/
```

Entity Graphs

When fetching entities with relationships (@OneToMany, @ManyToOne), JPA uses lazy loading by default. This can cause the N+1 query problem—one query for the main entity, followed by separate queries for each related entity.

Using @EntityGraph, we can specify which related entities should be eagerly fetched in a single query. This reduces the number of queries, improves performance, and lowers latency in applications with complex relationships.

```java
public interface UserRepository extends JpaRepository<User, Long> {

    @EntityGraph(value = "User.detail", type = EntityGraph.EntityGraphType.LOAD)
    Optional<User> findById(Long id);

    // Or define inline entity graph (without named graph):
    @EntityGraph(attributePaths = {"address", "roles"})
    List<User> findAll();
}
```

```java
@Entity
@NamedEntityGraph(
    name = "User.detail",
    attributeNodes = {
        @NamedAttributeNode("address"),
        @NamedAttributeNode("roles")
    }
)
public class User {
    @Id
    private Long id;

    @OneToOne(fetch = FetchType.LAZY)
    private Address address;

    @ManyToMany(fetch = FetchType.LAZY)
    private Set<Role> roles;

    // getters/setters
}
```

Locking Mechanisms

In high-concurrency environments, simultaneous updates can cause data inconsistencies. Spring Data JPA offers locking strategies to prevent this:

- **Optimistic Locking:** Uses a version field to prevent lost updates without blocking. Best for read-heavy workloads.
- **Pessimistic Locking:** Blocks concurrent writes by locking records during a transaction.
- **Row-Level Locks:** Use SQL (e.g., FOR UPDATE) for manual locking in custom queries.

```java
@Entity
class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Long id;

    private String name;
    private int stockQuantity;

    @Version
    private int version; // Prevents lost updates
}

Product product = repo.findById(1L);
product.setStockQuantity(product.getStockQuantity() - 1); // Decrease stock by 1
repo.save(product);

/* WITH @Version:
   Hibernate checks the version field before updating. If another transaction
   modified the record, it throws an OptimisticLockException.

   WITHOUT @Version: Concurrent updates may overwrite each other without
   detection. */

@Lock(LockModeType.PESSIMISTIC_WRITE)
@Query("SELECT p FROM Product p WHERE p.id = :id")
Product findByIdForUpdate(@Param("id") Long id);

/* WITHOUT Locking:
   Multiple transactions can modify the same record simultaneously, leading to
   inconsistent data. */
                                        /* WITH @Lock:
   @Lock(LockModeType.PESSIMISTIC_WRITE)
   Prevents other transactions from modifying the row until the current one is finished. */
```

QueryHints

Spring Data JPA provides the @QueryHints annotation to fine-tune query execution by passing metadata hints to the underlying JPA provider (such as Hibernate). These hints do not alter the query logic itself but can improve performance, reduce memory usage, and optimize caching behavior.

```java
@Query("SELECT u FROM User u WHERE u.status = 'ACTIVE'")
@QueryHints({
    // Marks query as read-only
    @QueryHint(name = "org.hibernate.readOnly", value = "true"),
    // Fetches records in batches of 100
    @QueryHint(name = "org.hibernate.fetchSize", value = "100"),
    // Enables second-level caching
    @QueryHint(name = "org.hibernate.cacheable", value = "true"),
    // Uses cache if available
    @QueryHint(name = "jakarta.persistence.cache.retrieveMode", value = "USE"),
    // Stores results in cache
    @QueryHint(name = "jakarta.persistence.cache.storeMode", value = "USE"),
    // Sets query timeout to 2000ms
    @QueryHint(name = "jakarta.persistence.query.timeout", value = "2000")
})
List<User> findActiveUsers();
```

Projections and DTOs

Fetching entire entities when only a few fields are needed can lead to performance issues, high memory usage, and slow API responses. By using projections and DTOs (Data Transfer Objects), we can retrieve only the required fields, reducing database load and improving query efficiency.
Spring Data JPA supports interface-based projections, which allow query results to be mapped directly into DTOs without loading the entire entity. This approach is particularly useful for optimizing REST APIs and avoiding unnecessary object hydration.

```java
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String fullName;
    private String email;
    private String password;
    private String address;
    private LocalDate birthDate;

    // Getters and setters...
}
```

```java
public interface UserRepository extends JpaRepository<User, Long> {
    // This will return only the fields defined in UserSummary
   //Note: findAllBy() is just findAll() with projection.
    List<UserSummary> findAllBy();
}
```

```java
public interface UserSummary {
    String getFullName();
    String getEmail();
}
```

Projections and DTOs

```java
package com.chakir.dto;

import java.time.LocalDate;

public class UserDTO {

    private String fullName;
    private String email;
    private LocalDate createdAt;

    public UserDTO(String fullName, String email, LocalDate createdAt) {
        this.fullName = fullName;
        this.email = email;
        this.createdAt = createdAt;
    }

    // Getters
}
```

```java
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("""
        SELECT new com.chakir.dto.UserDTO(u.fullName, u.email, u.createdAt)
        FROM User u
        WHERE u.createdAt > :date
    """)
    List<UserDTO> findUsersRegisteredAfter(@Param("date") LocalDate date);
}
```

Auditing

Spring Boot offers built-in support for automatic auditing to track entity changes such as timestamps and user details. This is especially useful for logging, compliance, and debugging purposes.

🔍 Key Concepts:

- @EnableJpaAuditing: Enables JPA Auditing in a Spring Boot application.
- @EntityListeners(AuditingEntityListener.class): Hooks into JPA lifecycle events to trigger auditing behavior.
- Auditing Annotations:
  - @CreatedDate: Automatically stores the timestamp when the entity is created.
  - @LastModifiedDate: Stores the timestamp when the entity is last updated.
  - @CreatedBy: Stores the user who created the entity.
  - @LastModifiedBy: Stores the user who last modified the entity.

```java
import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.*;
import org.springframework.data.jpa.domain.support.AuditingEntityListener;

import java.time.ZonedDateTime;

@MappedSuperclass // Allows other entities to inherit these fields
@EntityListeners(AuditingEntityListener.class) // Enables auditing
@Getter
@Setter
public abstract class Auditable {

    @CreatedDate
    private ZonedDateTime createdAt; // Timestamp when entity was created

    @LastModifiedDate
    private ZonedDateTime updatedAt; // Timestamp when entity was last modified

    @CreatedBy
    private String createdBy; // User who created the entity

    @LastModifiedBy
    private String updatedBy; // User who last modified the entity
}

@Entity
public class User extends Auditable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;
}

/*
WITH Auditing: Automatically tracks createdAt, updatedAt, createdBy, and updatedBy for every
entity. No need for manual timestamps in each entity. Works seamlessly with Spring Security to track
authenticated users.

WITHOUT Auditing:
1 - Developers must manually set timestamps and user information in every update.
2 - Higher risk of inconsistencies across entities.
*/
```

Criteria API

Instead of writing string-based JPQL queries, which are error-prone and vulnerable to SQL injection, the Criteria API allows developers to construct queries programmatically in a type-safe manner. This approach enhances maintainability, avoids hardcoded query strings, and enables dynamic query generation at runtime. It is especially useful for filtering data based on user input.

```java
@Service
public class UserService {
    @PersistenceContext
    private EntityManager entityManager;


    public List<User> findUsersNamedChakir() {
        // 1. Get CriteriaBuilder from EntityManager
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();

        // 2. Create a CriteriaQuery for User entities
        CriteriaQuery<User> query = cb.createQuery(User.class);

        // 3. Define the root of the query (FROM User u)
        Root<User> root = query.from(User.class);

        // 4. Add a WHERE condition: u.name = 'Chakir'
        query.select(root).where(cb.equal(root.get("name"), "Chakir"));

        // 5. Execute the query and return results
        List<User> results = entityManager.createQuery(query).getResultList();

        return results;
    }

    /*
    ✅ WITH Criteria API:
    - Type-safe: avoids typo errors at runtime.
    - SQL-injection safe: parameters are bound, not concatenated.
    - Dynamic query construction is easier and clean.

    ❌ WITHOUT Criteria API (e.g. using JPQL):
    - You might write "SELECT u FROM User u WHERE u.name = 'Chakir'"
    - Harder to construct dynamically.
    - Prone to runtime errors or injection if not careful.
    */
}
```

QueryDSL

QueryDSL is an advanced alternative to JPQL and the Criteria API, offering a fluent API for building queries in a more readable, maintainable, and type-safe manner. Unlike manually concatenated query strings, QueryDSL generates meta-model classes and enables dynamic chaining of conditions. This approach reduces query complexity and enhances code structure and clarity.

```java
@Service
public class UserService {

    @PersistenceContext
    private EntityManager entityManager;

    public List<User> findUsersNamedChakir() {
        JPAQueryFactory queryFactory = new JPAQueryFactory(entityManager);
        QUser user = QUser.user;

        return queryFactory
                .selectFrom(user)
                .where(user.name.eq("Chakir"))
                .fetch();
    }
}
```

THANK YOU

Like, Share & Follow

CHAKIR CHAIMAE