Spring Boot Data & Persistence

In diesem Modul

- JPA/EntityManager, Repository-Pattern
- JPQL, Fetch Joins und EntityGraph
- Projections/DTOs, Auditing, Transaktionen (Propagation/Isolation)
- Database Migrations (Flyway, Liquibase)
- Native Queries & Specification API
- NoSQL: MongoDB, Redis, Distributed Locks

JPA-Architektur

Spring Data legt eine Abstraktionsschicht über den JPA Provider (meist Hibernate).

1. **JPA (Java Persistence API):** Standard-Interfaces wie `EntityManager`.
2. **Hibernate:** Die wichtigste Implementierung.
3. **Spring Data JPA:** Abstraktionsschicht über Hibernate mit Repositories, die Boilerplate-Code reduziert.

Der Entity Manager

Auch wenn wir meistens Repositories nutzen, arbeitet im Hintergrund immer der `EntityManager`.

```
@PersistenceContext
private EntityManager em;

public User save(User user) {
    em.persist(user); // Objekt in den Persistence Context aufnehmen
    return user;
}
```

Der Persistence Context ist ein **First-Level Cache**. Änderungen an Managed Entities werden beim Transaktionsende automatisch in die DB geschrieben ("Dirty Checking").

Repository Pattern

Statt DAOs manuell zu schreiben, definieren wir Interfaces.

```
// Erbt CRUD-Methoden (save, findById, delete...)
public interface UserRepository extends JpaRepository<User, Long> {

    // Derived Query Methods (werden aus dem Methodennamen generiert)
    List<User> findByLastnameAndActiveTrue(String lastname);

    // JPQL Query
    @Query("SELECT u FROM User u WHERE u.email LIKE %:domain")
    List<User> findByEmailDomain(@Param("domain") String domain);
}
```

JPQL – Die Abfragesprache von JPA

- JPQL (*Java Persistence Query Language*) ist eine objektorientierte Abfragesprache, ähnlich zu SQL, aber operiert auf **Entities** und **ihren Attributen** statt auf Tabellen und Spalten.
- Der JPA Provider (z. B. Hibernate) übersetzt JPQL zur Laufzeit in vendor-spezifisches SQL.
- Vorteil: Queries bleiben portabel und eng an das Domain-Modell gekoppelt.

Grundsyntax einer JPQL-Query

```
@Query("SELECT u FROM User u WHERE u.active = true")
List<User> findActiveUsers();
```

Parametrisierung

```
@Query("SELECT u FROM User u WHERE u.email = :email")
User findByEmail(@Param("email") String email);
```

Inner Join

```
@Query("SELECT o FROM Order o JOIN o.customer c WHERE c.status = 'PREMIUM'")
List<Order> findOrdersOfPremiumCustomers();
```

Fetch Join

```
@Query("SELECT u FROM User u JOIN FETCH u.roles")
List<User> findAllWithRoles();
```

Eager Loading

Das N+1-Problem

Das N+1-Problem beschreibt, was häufig beim Iterieren über Entities passiert, wenn sie selbst eine 1:N-Relation haben.
Man lädt zum Beispiel 100 User in einer Query. Dann greift man auf `user.getAddresses()` zu, welches standardmäßig lazy geschieht.

- Für jeden der 100 User wird ein neues SELECT gefeuert.
- 1 + 100 = 101 Queries.
- Das ist ein Performance-Killer.

Lösung 1: @EntityGraph

Deklaratives Eager-Loading im Repository.

```
@EntityGraph(attributePaths = {"addresses"})
List<User> findAll();
```

Lösung 2: JPQL Fetch Join

Lädt ebenfalls den ganzen Graph.

```
@Query("SELECT u FROM User u JOIN FETCH u.addresses")
List<User> findAllWithAddresses();
```

Projections

Projections mit DTOs

- Projektionen laden nur Teile der Entities.
- Dafür schreibt man neue Klassen, idealerweise als Record, welche die Projektion aufnehmen.
- Dafür gibt es drei Optionen:
  - Interface Projection
  - Class Projection
  - JPQL Projection

Interface Projection

Spring generiert zur Laufzeit einen Proxy.

```
public interface UserView {
    String getUsername();
    // Open Projection (SpEL)
    @Value("#{target.firstname + ' ' + target.lastname}")
    String getFullName();
}
```

Class Projection (Records)

Type-safe und performant (selektiert nur benötigte Spalten im SQL).

```
public record UserDto(String username, String email) {}
// Im Repo:
List<UserDto> findByActiveTrue();
```

JPQL-Projektion

```
@Query("""
    SELECT new com.example.UserSummary(u.username, u.email)
    FROM User u
    WHERE u.active = true
""")
List<UserSummary> findActiveUserSummaries();
```

Auditing

Automatisches Tracking von Änderungen.

1. `@EnableJpaAuditing` in der Config.
2. Entity anpassen:

```
@EntityListeners(AuditingEntityListener.class)
public class User {
    @CreatedDate
    private LocalDateTime createdAt;

    @LastModifiedBy
    private String lastModifiedBy;
}
```

Transaktionsmanagement

Basics

In Spring markiert `@Transactional` Methoden, die atomar ausgeführt werden sollen.

- Default: Rollback nur bei `RuntimeException` (unchecked).
- Checked Exceptions (z.B. `IOException`) lösen standardmäßig **keinen** Rollback aus!
  -> `@Transactional(rollbackFor = Exception.class)`

Propagation

Wie verhalten sich Transaktionen bei verschachtelten Service-Aufrufen?

- **REQUIRED (Default):** Nutze vorhandene TX, sonst neue starten.
- **REQUIRES_NEW:** Starte *immer* eine neue TX (pausiere die alte). Wichtig für Logs, die trotz Rollback geschrieben werden sollen.
- **SUPPORTS:** Laufe in TX wenn da, sonst ohne.
- **MANDATORY:** Wirf Exception, wenn keine TX da ist.

```
@Service
public class OrderService {

    private final OrderRepository orders;
    private final AuditService audit;
    private final PaymentService payments;

    @Transactional // REQUIRED als default: gemeinsamer Commit/Rollback
    public void place(Order order) {
        orders.save(order);
        payments.charge(order); // Exception → alles rollt zurück
        audit.logOrder(order);
    }

    @Transactional
    public void placeWithAuditSafe(Order order) {
        orders.save(order);
```

Isolation Levels

- **READ_COMMITTED:** Standard. Verhindert Dirty Reads.
- **REPEATABLE_READ:** Verhindert Non-Repeatable Reads.
- **SERIALIZABLE:** Sperrt Tabellen/Rows aggressiv. Sicher, aber langsam.

```java
@Service
public class InventoryService {

    private final ProductRepository products;

    // Sicher gegen Non-Repeatable Reads (z.B. doppelte Reservierung)
    @Transactional(isolation = Isolation.REPEATABLE_READ)
    public void reserve(String sku, int qty) {
        Product p = products.findBySkuForUpdate(sku) // Query mit PESSIMISTIC_WRITE
            .orElseThrow();
        if (p.getStock() < qty) throw new IllegalStateException("Not enough stock");
        p.decreaseStock(qty);
    }

    // Strenger: konsistente Prüfung über mehrere Zeilen (Phantoms vermeiden)
    // Beispiel: Gesamtsumme aller Reservierungen darf den Bestand nicht übersteigen.
    @Transactional(isolation = Isolation.SERIALIZABLE)
    public void placeBulkOrder(String sku, int requested) {
        int alreadyReserved = products.sumReservations(sku); // SELECT SUM(...) FROM reservations WHERE sku=?
        Product p = products.findBySkuForUpdate(sku).orElseThrow();
        if (alreadyReserved + requested > p.getStock()) {
            throw new IllegalStateException("Overbooking prevented");
        }
        products.insertReservation(sku, requested); // eigene Tabelle/Row
    }
}

public interface ProductRepository extends JpaRepository<Product, Long> {
    @Lock(LockModeType.PESSIMISTIC_WRITE)
    @Query("select p from Product p where p.sku = :sku")
    Optional<Product> findBySkuForUpdate(@Param("sku") String sku);

    @Query("select coalesce(sum(r.qty),0) from Reservation r where r.sku = :sku")
    int sumReservations(@Param("sku") String sku);

    @Modifying
    @Query("insert into Reservation(sku, qty) values (:sku, :qty)")
    void insertReservation(@Param("sku") String sku, @Param("qty") int qty);
}
```

Database Migrations

Warum Database Migrations?

- **Versionierung:** Schema-Änderungen sind nachvollziehbar (Git).
- **Reproduzierbarkeit:** Gleiche Migration auf Dev, Test, Prod.
- **Team-Arbeit:** Konflikte bei Schema-Änderungen werden sichtbar.
- **Rollback:** (Eingeschränkt) Zurückrollen von Änderungen möglich.

Tools: Flyway, Liquibase

Flyway vs. Liquibase

Feature	Flyway	Liquibase
Format	SQL, Java	XML, YAML, JSON, SQL
Lernkurve	Einfach	Komplexer
Rollback	Manuell (Pro: automatisch)	Automatisch generierbar
DB-Agnostisch	Nein (SQL-basiert)	Ja (abstraktes Format)
Spring Boot	✅ Auto-Config	✅ Auto-Config

Flyway Setup

Dependency: `org.flywaydb:flyway-core`

Struktur:

```
src/main/resources/
└── db/migration/
    ├── V1__create_users_table.sql
    ├── V2__add_email_column.sql
    └── V3__create_orders_table.sql
```

Namenskonvention: `V{version}__{description}.sql`

Flyway Migration Beispiel

V1__create_users_table.sql:

```sql
CREATE TABLE users (
    id BIGSERIAL PRIMARY KEY,
    username VARCHAR(100) NOT NULL UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_users_username ON users(username);
```

V2__add_email_column.sql:

```sql
ALTER TABLE users ADD COLUMN email VARCHAR(255);
UPDATE users SET email = username || '@example.com' WHERE email IS NULL;
ALTER TABLE users ALTER COLUMN email SET NOT NULL;
```

Flyway Konfiguration

```yaml
spring:
  flyway:
    enabled: true
    locations: classpath:db/migration
    baseline-on-migrate: true  # Für bestehende DBs
    validate-on-migrate: true  # Prüft Checksummen

    # Für verschiedene Umgebungen
    # placeholders:
    #   schema: ${DB_SCHEMA:public}
```

Liquibase Setup

Dependency: `org.liquibase:liquibase-core`

Struktur:

```
src/main/resources/
└── db/changelog/
    ├── db.changelog-master.yaml
    ├── changes/
    │   ├── 001-create-users.yaml
    │   └── 002-add-orders.yaml
```

Liquibase Changelog Beispiel

db.changelog-master.yaml:

```yaml
databaseChangeLog:
  - include:
      file: db/changelog/changes/001-create-users.yaml
  - include:
      file: db/changelog/changes/002-add-orders.yaml
```

Liquibase Change Set

001-create-users.yaml:

```yaml
databaseChangeLog:
  - changeSet:
      id: 1
      author: aerben
      changes:
        - createTable:
            tableName: users
            columns:
              - column:
                  name: id
                  type: BIGINT
                  autoIncrement: true
                  constraints:
                    primaryKey: true
              - column:
                  name: username
                  type: VARCHAR(100)
                  constraints:
                    nullable: false
                    unique: true
      rollback:
        - dropTable:
            tableName: users
```

Native Queries

@Query mit nativeQuery = true

Manchmal reicht JPQL nicht aus – dann braucht man echtes SQL.

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    @Query(value = """
        SELECT * FROM products p
        WHERE p.price < :maxPrice
        AND p.category_id IN (
            SELECT c.id FROM categories c WHERE c.active = true
        )
        ORDER BY p.created_at DESC
        LIMIT :limit
        """, nativeQuery = true)
    List<Product> findCheapProductsInActiveCategories(
        @Param("maxPrice") BigDecimal maxPrice,
        @Param("limit") int limit
    );
}
```

Native Query: Wann verwenden?

Szenario	Empfehlung
DB-spezifische Funktionen (z.B. `JSONB`, `ARRAY`)	Native Query
Window Functions (`ROW_NUMBER`, `RANK`)	Native Query
Komplexe Subqueries	Native Query
Einfache CRUD	JPQL oder Derived Query
Portabilität wichtig	JPQL

Achtung: Native Queries umgehen den Entity-Cache!

Native Query mit Projektion

```
public interface OrderStatistics {
    String getStatus();
    Long getCount();
    BigDecimal getTotalAmount();
}

public interface OrderRepository extends JpaRepository<Order, Long> {

    @Query(value = """
        SELECT status, COUNT(*) as count, SUM(amount) as totalAmount
        FROM orders
        WHERE created_at >= :since
        GROUP BY status
        """, nativeQuery = true)
    List<OrderStatistics> getOrderStatistics(@Param("since") LocalDate since);
}
```

Specification API

Dynamische Queries mit Specifications

Die Specification API ermöglicht **dynamische, typsichere Queries** zur Laufzeit.

Problem:

```
// So nicht! Kombinatorische Explosion von Methoden
findByStatusAndCategoryAndPriceGreaterThan(...)
findByStatusAndCategory(...)
findByStatus(...)
findByCategoryAndPriceGreaterThan(...)
```

Lösung: Specifications kombinieren!

Repository erweitern

```
public interface ProductRepository extends
    JpaRepository<Product, Long>,
    JpaSpecificationExecutor<Product> {  // <-- Hinzufügen

    // Keine zusätzlichen Methoden nötig
}
```

Specifications definieren

```
public class ProductSpecifications {

    public static Specification<Product> hasStatus(ProductStatus status) {
        return (root, query, cb) ->
            status == null ? null : cb.equal(root.get("status"), status);
    }

    public static Specification<Product> inCategory(Long categoryId) {
        return (root, query, cb) ->
            categoryId == null ? null : cb.equal(root.get("category").get("id"), categoryId);
    }

    public static Specification<Product> priceBetween(BigDecimal min, BigDecimal max) {
        return (root, query, cb) -> {
            if (min == null && max == null) return null;
            if (min == null) return cb.lessThanOrEqualTo(root.get("price"), max);
            if (max == null) return cb.greaterThanOrEqualTo(root.get("price"), min);
            return cb.between(root.get("price"), min, max);
        };
    }
}
```

Specifications kombinieren

```
@Service
public class ProductService {

    public List<Product> search(ProductSearchCriteria criteria) {
        Specification<Product> spec = Specification
            .where(ProductSpecifications.hasStatus(criteria.getStatus()))
            .and(ProductSpecifications.inCategory(criteria.getCategoryId()))
            .and(ProductSpecifications.priceBetween(criteria.getMinPrice(), criteria.getMaxPrice()));

        return productRepository.findAll(spec, Sort.by("name"));
    }
}
```

Die Specifications werden nur angewendet, wenn der Parameter **nicht null** ist!

Advanced-Themen zu NoSQL

MongoDB: Optimistic Locking

Verhindert "Lost Updates" in verteilten Systemen ohne harte DB-Locks.

```
@Document
public class Product {
    @Id String id;
    @Version Long version; // Spring Data prüft und inkrementiert dies
}
```

Wenn zwei User gleichzeitig speichern, gewinnt der erste. Der zweite bekommt eine `OptimisticLockingFailureException`.

Redis als Cache

Caching beschleunigt Lesezugriffe dramatisch.

```
@Service
public class PricingService {

    @Cacheable(value = "prices", key = "#productId")
    public BigDecimal getPrice(String productId) {
        // Teure Berechnung oder DB-Call
        return calculatePrice(productId);
    }

    @CacheEvict(value = "prices", key = "#productId")
    public void updatePrice(String productId, BigDecimal newPrice) {
        // Update Logik
    }
}
```

Distributed Locks (ShedLock)

Szenario: Eine `@Scheduled` Methode soll in einem Cluster (3 Instanzen) nur **einmal** ausgeführt werden.

Lösung: ShedLock (nutzt DB oder Redis als Lock-Provider).

```
@Scheduled(cron = "0 0 * * * *")
@SchedulerLock(name = "dailyReport", lockAtMostFor = "10m")
public void generateDailyReport() {
    // Läuft garantiert nur auf einer Instanz gleichzeitig
}
```