Lernskript: Spring Data JPA

Spring Data JPA ist eine Abstraktion, die den Aufwand für die Datenbankzugriffs-Schicht (Repositories) in Java-Anwendungen drastisch reduziert. Man schreibt fast nie wieder SQL oder Implementierungs-Code für CRUD-Operationen.

Teil 1: Die grundlegende Analogie - Das "Warum"

- **Konzept:** Anstatt Datenbankoperationen manuell mit JDBC zu programmieren, definiert man nur noch ein **Interface** (einen "Wunschzettel" oder "Briefkasten"). Spring Data JPA agiert wie ein **intelligentes Postamt**, das die Methodennamen auf diesem Interface liest, ihre Absicht versteht und zur Laufzeit automatisch die komplette Implementierung inklusive des SQL-Codes generiert.
- **Die Ebenen der Abstraktion:**
    1. **JDBC:** Manuelle, unterste Ebene.
    2. **JPA/Hibernate:** Ein "Assistent", der Java-Objekte in SQL übersetzt (Object-Relational Mapping - ORM).
    3. **Spring Data JPA:** Eine weitere Schicht darüber, die sogar das Schreiben der meisten JPA-Operationen automatisiert.

Teil 2: Praktische Anwendungsfälle - Das "Wofür"

Use Case 1: Die `@Entity` - Der Bauplan für die Tabelle

- **Zweck:** Man markiert eine normale Java-Klasse (POJO), um sie mit einer Datenbanktabelle zu verknüpfen.
- **Wichtige Annotationen (aus `jakarta.persistence`):**
    - `@Entity`: Markiert die Klasse als Datenbank-Entität.
    - `@Id`: Markiert das Feld, das als Primärschlüssel dient.
    - `@GeneratedValue`: Konfiguriert die automatische Generierung des Primärschlüssels (z.B. durch Auto-Inkrement).
- **Code-Beispiel:**

```
@Entity
public class Benutzer {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
    // ...
}
```

Use Case 2: Das Repository - Der magische Briefkasten

```
Zweck: Man definiert die Datenbankoperationen, die man für eine Entität benötigt.

Lösung: Man erstellt ein Interface, das von JpaRepository erbt.

Code-Beispiel:


public interface BenutzerRepository extends JpaRepository<Benutzer, Long> {
    // JpaRepository<Entity-Typ, ID-Typ>
}



Ergebnis: Durch diese eine Zeile erhält man automatisch Methoden wie save(),
findById(), findAll(), deleteById() und viele mehr.
```

Use Case 3: Eigene Abfragen durch Methodennamen

```
Zweck: Definition von spezifischen Suchanfragen ohne SQL zu schreiben.

Lösung: Man fügt dem Repository-Interface Methoden hinzu, deren Namen einer festen
Konvention folgen.

Code-Beispiel:

public interface BenutzerRepository extends JpaRepository<Benutzer, Long> {
    // Generiert: SELECT b FROM Benutzer b WHERE b.name = ?
    Optional<Benutzer> findByName(String name);

    // Generiert: SELECT b FROM Benutzer b WHERE b.abteilung = ? AND b.gehalt > ?
    List<Benutzer> findByAbteilungAndGehaltGreaterThan(String abteilung, double
gehalt);
}
```

Teil 3: Vertiefung (JavaMasta's Profi-Tipps)

```
JPA vs. Hibernate: JPA ist die Spezifikation (das Regelwerk). Hibernate ist die
populärste Implementierung dieses Regelwerks. Spring Data JPA benutzt
standardmäßig Hibernate "unter der Haube".

Eigene Abfragen mit @Query: Für sehr komplexe Abfragen, die nicht durch
Methodennamen abgebildet werden können, kann man die SQL- (oder JPQL-) Abfrage
direkt in der @Query-Annotation an die Methode schreiben.
```

```
Transaktionen (@Transactional): Jede schreibende Datenbankoperation (save, delete)
muss in einer Transaktion ablaufen. In Spring wird dies typischerweise durch die
Annotation @Transactional an der Service-Methode sichergestellt, die die
Repository-Methode aufruft. Spring kümmert sich dann automatisch um Commit (bei
Erfolg) und Rollback (bei Fehlern).
```