Softwareentwicklung (SW)

Persistenz Teil 2 : Relations und Mappings N:M

Prof. Dr. Alixandre Santana
alixandre.santana@oth-regensburg.de

Wintersemester
2024/2025

Lernziele

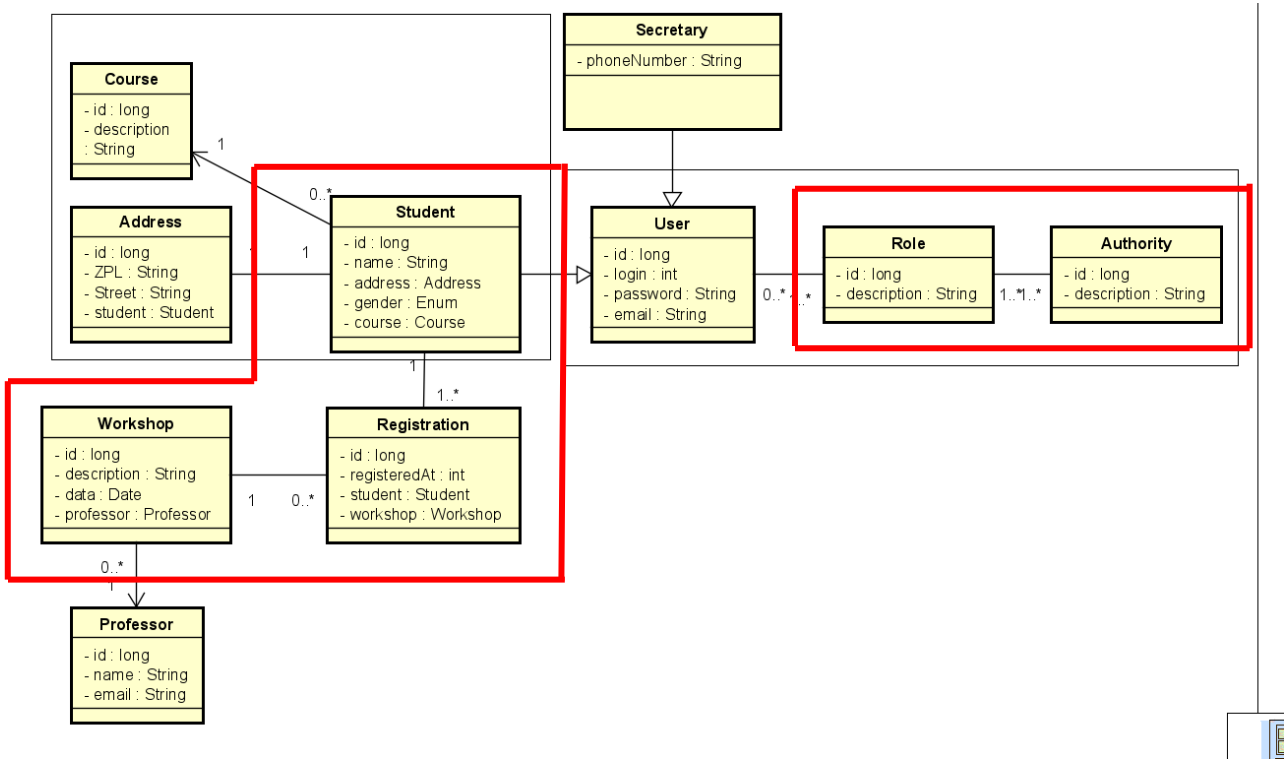
- Die Beziehungen N:M und Auswirkungen zu beschreiben
- Die Strategien für N:M Bezienhungen zu verstehen
- N:N-Beziehungen zu implementieren

Agenda

1. Die Beziehungen N:M
2. Die Strategien für N:M Bezienhungen

1.1 N:M Relations

- Eine Beziehung ist eine Verbindung zwischen zwei Arten von Entitäten.
- Im Falle einer Many-To-Many Beziehung können beide Seiten mehreren Instanzen der jeweils anderen Seite Bericht erstatten.



1.1 N:M Relations

- Nehmen wir das Beispiel der Studenten, die die Kurse bewerten, die ihnen gefallen.
- Ein Student kann viele Kurse mögen, und Ein Kurs hat mehrere Studenten, die ihn mögen:

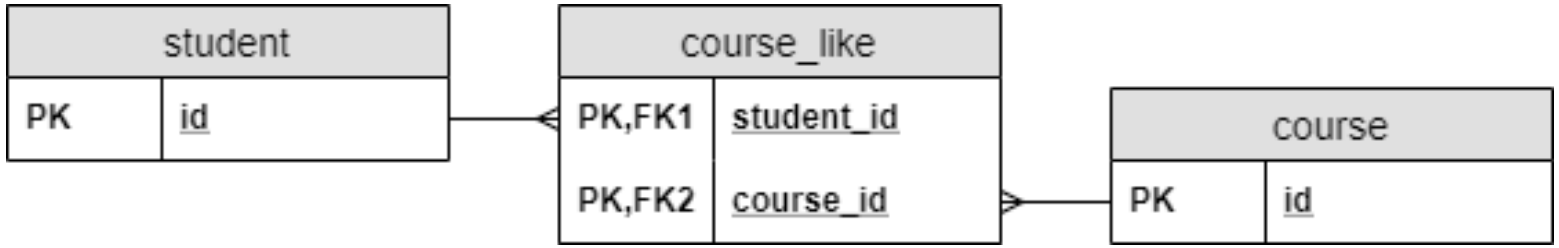
Sehen Sie: https://www.baeldung.com/jpa-many-to-many

Agenda

1. Die Beziehungen N:M
2. Die Strategien für Modelierung der N:M Bezienhungen

2.1 N:M Relations mit Join Table

- Wir können in RDBMS Beziehungen mit Fremdschlüsseln erstellen.
- Da beide Seiten in der Lage sein sollten, aufeinander zu verweisen, müssen wir eine separate Tabelle erstellen, um die Fremdschlüssel zu speichern:



- Eine solche Tabelle wird als **Join-Tabelle** bezeichnet. In einer Join-Tabelle ist die Kombination der Fremdschlüssel der zusammengesetzte Primärschlüssel.

Sehen Sie: https://www.baeldung.com/jpa-many-to-many

2.1 Join Table mit Spring Data/JPA

```
@Entity
class Student {
@Id
 Long id;
@ManyToMany
 @JoinTable( name = "course_like",
joinColumns = @JoinColumn(name =
"student_id"), inverseJoinColumns =
@JoinColumn(name = "course_id"))
Set<Course> likedCourses;

}
```

```
@Entity
class Course {

 @Id
 Long id;

@ManyToMany(mappedBy = "likedCourses")
 Set<Student> likes;

}
```

- Wir sollten in beiden Klassen eine Collection einbinden, die die Elemente der anderen enthält.
- Außerdem sollten wir den Beziehungstyp konfigurieren. Daher markieren wir die Collections mit @ManyToMany -Anmerkungen:

Sehen Sie: https://www.baeldung.com/jpa-many-to-many

2.1 Andere Beispiel: User X Role mit Join Table

```
@Entity
@Table(name="user")
public class User implements Serializable{
..
@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(
name="userrole",
joinColumns = @JoinColumn(name="iduser"),
inverseJoinColumns =
@JoinColumn(name="idrole")
)
private List<Role> roles = new
ArrayList<Role>();
```

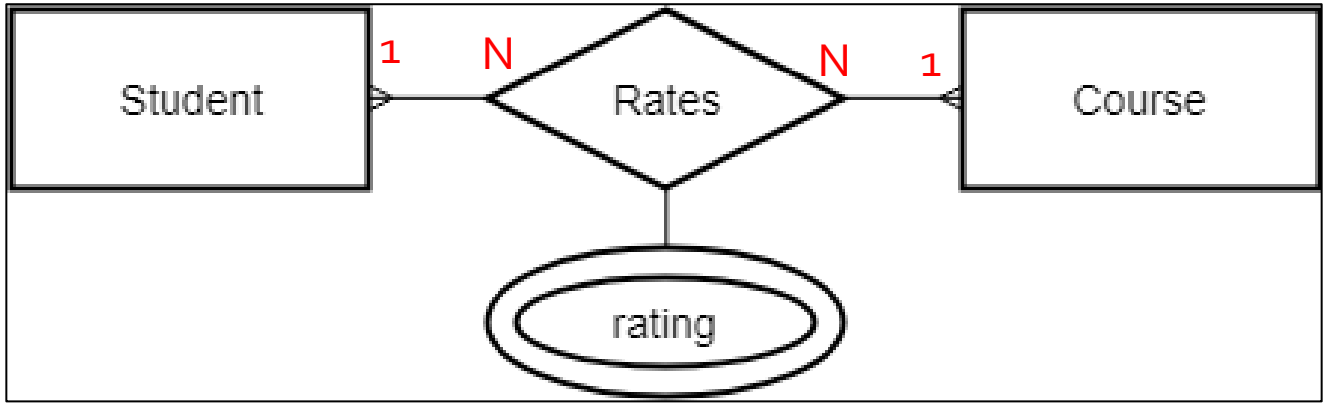
```
@Entity
@Table(name="role")
public class Role implements
Serializable {

..


@ManyToMany(mappedBy = "roles")
private Collection<User> users;
```

- Wir sollten in beiden Klassen eine Collection einbinden, die die Elemente der anderen enthält.
- Außerdem sollten wir den Beziehungstyp konfigurieren. Daher markieren wir die Collections mit @ManyToMany -Anmerkungen:

Sehen Sie: https://www.baeldung.com/jpa-many-to-many

2.2 N:M -Modeling Relationship Attributes

- Ein Student kann beliebig viele Kurse bewerten und ein Kurs kann von vielen Studenten bewertet werden. N:M schon wieder!
- Was dieses Beispiel etwas komplizierter macht, ist die Tatsache, dass wir die Bewertung speichern müssen, die der Student für den Kurs abgegeben hat.



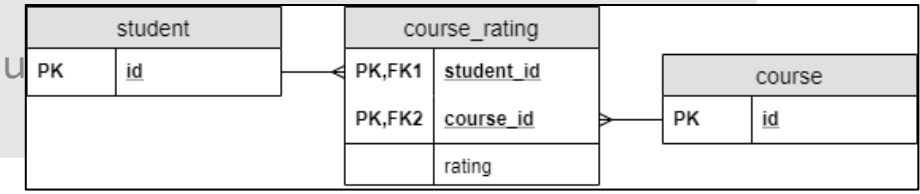
Sehen Sie: https://www.baeldung.com/jpa-many-to-many

2.2.1 N:M - Composite Key mit Spring Data/JPA

- Da unser Primärschlüssel ein zusammengesetzter Schlüssel ist, müssen wir eine neue Klasse erstellen (CourseRatingKey), die die verschiedenen Teile des Schlüssels enthält:

```
@Embeddable
class CourseRatingKey implements Serializable {

   @Column(name = "student_id")
   Long studentId;

   @Column(name = "course_id")
   Long courseId;

   // standard constructors, getters, and setters
   // hashcode and equals implementation
}
```

```
@Entity
class CourseRating {

@EmbeddedId CourseRatingKey id;

@ManyToOne
@MapsId("studentId") @JoinColumn(name =
"student_id")
Student student;
@ManyToOne
@MapsId("courseId") @JoinColumn(name =
"course_id")
Course course;
int rating;
// standard constru
}
```



Sehen Sie: https://www.baeldung.com/jpa-many-to-many

2.2.1 N:M - Composite Key mit Spring Data/JPA

- Wir haben @EmbeddedId verwendet, um den Primärschlüssel zu markieren, der eine Instanz der CourseRatingKey-Klasse ist.
- Wir haben die Felder „Student“ und „Kurs“ mit @MapsId markiert.
- @MapsId bedeutet, dass wir diese Felder mit einem Teil des Schlüssels verknüpfen und sie die Fremdschlüssel einer N:1-Beziehung sind.
- Wir haben die Beziehungen zu den Klassen „Student“ und „Course“ als zwei @ManyToOne konfiguriert.

```
@Entity
class CourseRating {

@EmbeddedId CourseRatingKey id;

@ManyToOne
@MapsId("studentId")
@JoinColumn(name = "student_id")
Student student;
@ManyToOne
@MapsId("courseId")
@JoinColumn(name = "course_id")
Course course;
int rating;
// standard constructors, getters, and setters
}
```

Sehen Sie: https://www.baeldung.com/jpa-many-to-many

2.2.1 N:M - Composite Key mit Spring Data/JPA

```
@Entity
class Student {

   // ...

   @OneToMany(mappedBy = "student")
   Set<CourseRating> ratings;

   // ...
}
```

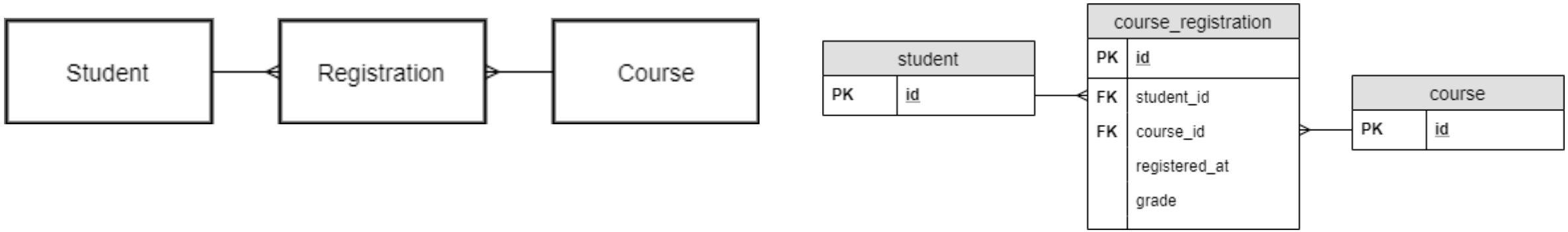
```
@Entity
class Course {

// ...

   @OneToMany(mappedBy = "course")
   Set<CourseRating> ratings;

   // ...
}
```

- Wir konnten dies tun, weil wir mit der neuen Entität die N:M-Beziehung strukturell in zwei N:1-Beziehungen zerlegt haben.

Sehen Sie: https://www.baeldung.com/jpa-many-to-many

2.2.2 N:M - with a New Entity

- In einigen realen Situationen kann derselbe Student denselben Kurs mehrmals belegen. In diesem Fall funktioniert der „Composite Key“ als Primärschlüssel nicht.
- In diesem Fall gibt es mehrere Verbindungen zwischen denselben Student-Kurs-Paaren oder mehreren Zeilen mit denselben „student_id“ – „course_id“ –Paaren.
- Daher können wir eine Entität einführen, die die Attribute der Registrierung enthält:



- Da es eine Entität ist, verfügt es über einen eigenen Primärschlüssel.

Sehen Sie: https://www.baeldung.com/jpa-many-to-many

2.2.2 N:M - with a New Entity mit Spring JPA – Beispiel von uns

```java
@Entity
@Table(name="registration")
public class Registration implements Serializable {

// ...

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

@ManyToOne
@JoinColumn(name = "student_id", referencedColumnName =
"id")
Student student;


@ManyToOne
@JoinColumn(name = "course_id", referencedColumnName = "id")
Workshop workshop;


@DateTimeFormat(pattern="yyyy-MM-dd")
LocalDate registeredAt;

  // ...
}
```

```java
@Entity
@Table(name="student")
public class Student implements Serializable{

@Id
@GeneratedValue (strategy = GenerationType.IDENTITY)
Long id;
@NotBlank(message = "Name is mandatory")
private String name;

 // ...
}
```

```java
@Entity
@Table(name="workshop")
public class Workshop implements Serializable{
@Id
@GeneratedValue (strategy = GenerationType.IDENTITY)
Long id;
@NotBlank(message = "Name is mandatory")
private String description;
  // ...
}
```

Review

Wir haben drei Möglichkeiten der Modellierung für N:M-Beziehungen gesehen::

- Join Table without Attributes
- Join Table With Composite-Key and Attributes
- Join Table with Own-Key and Attributes

Alle drei haben unterschiedliche Vor- und Nachteile, wenn es um diese Aspekte geht:

- Klarheit des Codes
- DB-Klarheit
- Fähigkeit, der Beziehung Attribute zuzuordnen
- Wie viele Entitäten können wir mit der Beziehung verknüpfen
- Unterstützung für mehrere Verbindungen zwischen denselben Entitäten

Sehen Sie: https://www.baeldung.com/jpa-many-to-many