



Hibernate



Prowadzący

Bartosz Andreatto
(bandreatto@gmail.com)

programista – Bank Pekao S.A.
<https://pl.linkedin.com/in/bartosz-andreatto-02b03413a>



ORM i JPA

Object-Relational Mapping – mapowanie obiektowo-relacyjne

Sposób odwzorowania obiektowej architektury systemu informatycznego na bazę danych (lub inny element systemu) o relacyjnym charakterze.

Implementacja takiego odwzorowania stosowana jest m.in. w przypadku, gdy tworzony system oparty jest na **podejściu obiektowym**, a **system bazy danych operuje na relacjach**.

Java Persistence API – oficjalny **standard mapowania obiektowo-relacyjnego** (ORM) firmy Sun Microsystems dla języka programowania Java.



Hibernate

Hibernate jest **frameworkiem implementującym standard JPA**. Podobnie jak JDBC API, **pozwala on na komunikację z bazą danych na poziomie aplikacji napisanej w języku Java**, ale zwalnia programistę z bezpośredniego korzystania z JDBC oraz, **w niektórych przypadkach, zwalnia programistę z obowiązku pisania zapytań SQL**. Robi to za pomocą dodatkowej warstwy abstrakcji.

Ponadto, **Hibernate wykorzystuje technikę ORM** (Object-relational mapping), która pozwala na **reprezentację tabeli bazodanowej za pomocą obiektu w języku Java**.

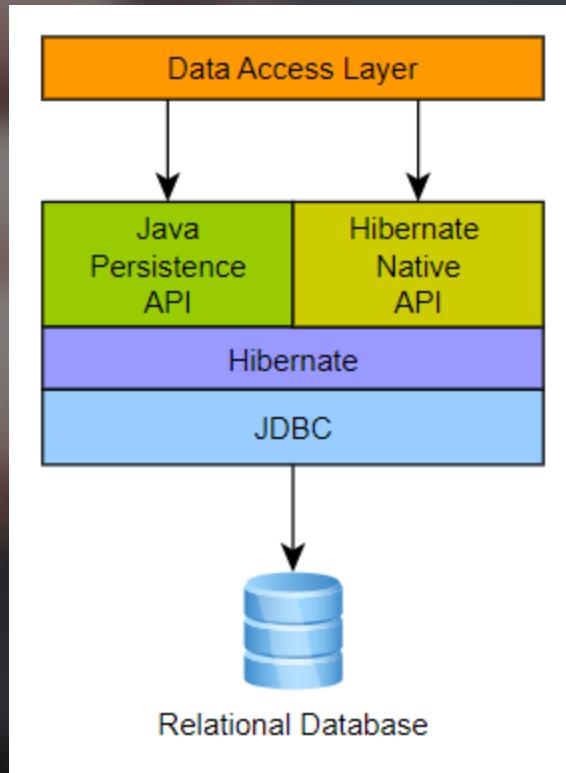


Hibernate

- **Jedna z dostępnych implementacji standardu JPA**, oferuje jednak więcej funkcji niż zakłada standard.
- Mapowanie między obiektami a rekordami w bazie danych jest **konfigurowalne za pomocą specjalnych adnotacji lub plików XML** (.hbm.xml).
- Dokumentacja:
http://docs.jboss.org/hibernate/orm/5.3/userguide/html_single/Hibernate_User_Guide.html



Architektura





Zależność maven

W celu wykorzystania Hibernate w projekcie, **w zależnościach projektu dodajemy:**

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.18.Final</version> <!-- Nowsza wersja może być dostępna -->
</dependency>
```



Konfiguracja

Aby móc skorzystać z framework'a Hibernate w aplikacji, **musimy stworzyć pewne pliki konfiguracyjne**. Jednym z najważniejszych komponentów jest plik **hibernate.cfg.xml**, który jest m.in. odpowiedzialny za konfigurację połączenia bazodanowego oraz zawiera informacje o klasach, które reprezentują tabele w bazie danych.



Konfiguracja

Przykładowa konfiguracja może wyglądać następująco:

```
1 <!DOCTYPE hibernate-configuration PUBLIC
2           "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
3           "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
4 <hibernate-configuration>
5   <session-factory>
6     <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
7     <property name="connection.url">jdbc:mysql://localhost:3306/sda</property>
8     <property name="connection.username">root</property>
9     <property name="connection.password">my-secret-pw</property>
10    <property name="dialect">org.hibernate.dialect.MySQL8Dialect</property>
11  </session-factory>
12 </hibernate-configuration>
```

W praktyce konfiguracja ta jest często bardziej skomplikowana, ale powyższy przykład pokazuje minimalne informacje, jakie Hibernate potrzebuje.



SessionFactory

Dzięki powyższemu plikowi konfiguracyjnemu **jesteśmy w stanie stworzyć obiekt SessionFactory, dzięki któremu stworzymy obiekt Session, który jest potrzebny do wykonywania jakichkolwiek operacji na bazie.** W celu stworzenia obiektu **SessionFactory** wykorzystujemy obiekt **Configuration**, np.:

```
SessionFactory sessionFactory = new Configuration()  
    .configure("hibernate.cfg.xml")  
    .buildSessionFactory();
```



Session

Dzięki obiektowi **SessionFactory**, tworzymy obiekt **Session**, który ma możliwość bezpośrednio wykonać operacje na bazie danych. Zgodnie z intencją obiekt ten **powinien być tworzony każdorazowo**, gdy niezbędne jest wykonanie operacji (jednej bądź kilku) bazodanowej. **Po ich wykonaniu sesja powinna zostać zamknięta.**

UWAGA: **Obiekt Session implementuje interfejs AutoCloseable**, więc tworzenie sesji możemy wykonywać z wykorzystaniem try-z-zasobem.

UWAGA: **Obiekt Session implementuje interfejs EntityManager**, który daje możliwość zarządzania encjami.



Transakcje

Session jest obiektem, który pozwala **tworzyć i zarządzać transakcjami**. W celu utworzenia transakcji wywołujemy metodę **getTransaction**, następnie po wykonaniu operacji na bazie, możemy wywołać metodę **commit** lub **rollback**.



Transakcje

```
final SessionFactory sessionFactory = new Configuration()
    .configure("hibernate.cfg.xml")
    .buildSessionFactory();

try (Session session = sessionFactory.openSession()) {
    Transaction transaction = session.beginTransaction();

    // wykonanie operacji bazodanowych

    if (success) {
        transaction.commit();
    } else {
        transaction.rollback();
    }
}
```



Modelowanie encji

W celu nadania możliwości zarządzania encjami obiektowi EntityManager, **musimy je odpowiednio oznaczyć**. Możemy to zrobić **poprzez konfigurację w pliku xml lub poprzez adnotacje**. Prezentacja przedstawia konfiguracje **wykonane na poziomie kodu**.



@Entity

W celu zdefiniowania encji bazodanowej należy **deklarację klasy oznać adnotacją @Entity**, np.

```
@Entity  
public class Movie {  
    // deklaracja pól klasy  
}
```

Adnotacja **@Entity pozwala dodatkowo w atrybucie name, zdefiniować nazwę tabeli**, w której Hibernate oczekuje wartości encji.

UWAGA: Każda encja musi być oznaczona adnotacją @Entity.



@Table

Adnotacja **@Table** pozwala, tak jak adnotacja @Entity, **zdefiniować nazwę tabeli, ale również wskazać katalog lub nazwę bazy danych**, w której się ona znajduje. **Umożliwia także zdefiniowanie dodatkowych indeksów i ograniczeń** (np. unikalność wartości kolumn lub grupy).

```
@Entity  
@Table(name = "movies")  
public class Movie {  
    // definicja tabeli  
}
```

UWAGA: Adnotacja @Table nie jest wymagana do poprawnego zdefiniowania encji.



@Column

Domyślnie nazwy **pól są bezpośrednio mapowane na nazwę kolumn** w tabeli bazodanowej. Adnotacja **@Column** umożliwia określenie konkretnej mapowanej **nazwy**, która będzie reprezentować nazwę kolumny bazodanowej, np.:

```
@Entity  
@Table(name = "movies")  
public class Movie {  
  
    private String name;  
  
    @Column(name = "movie_title")  
    private String title;  
  
    // pozostałe pola klasy  
}
```

Adnotacja ta umożliwia również określanie **długości kolumny, definiuje czy wartość może być nullem oraz, czy powinna być wymuszona unikalność.**

UWAGA: Encja bazodanowa powinna **definiować bezargumentowy konstruktor oraz gettery i settery dla wszystkich pól.**



@SecondaryTable

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "specialization")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Specialization {

    @Id
    private int id;

    private String type;
}
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;

@Entity
@Table(name = "teacher")
@SecondaryTable(name = "specialization")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Teacher {

    @Id
    private int id;

    @Embedded
    private Person person;

    @Column(table = "specialization")
    private String type;
}
```



@MappedSuperclass

Relacyjne bazy **danych łączą tabele na zasadzie wiązań, natomiast encje realizują w pełni obiektowy paradygmat**, w związku z tym wiele aspektów logiki biznesowej można rozwiązać za pomocą **dziedziczenia**, co nie jest rozwiązaniem zgodnym z koncepcją relacyjnych baz danych. Podstawową formą rozwiązania tego problemu jest wykorzystanie adnotacji **@MappedSuperclass**, która **umożliwia tworzenia klas bazowych, które nie będą mieć fizycznego odzwierciedlenia w tabelach**, a służyć będą do określania podstawy dla encji. **Wszystkie kolumny z klasy oznaczonej @MappedSuperclass będą zawierać się w klasie pochodnej, która jest właściwą realizacją tabeli.**



@MappedSuperclass

```
@MappedSuperclass
public abstract class BaseEntity {

    @Id
    private int id;
}

@Entity
public class ComputerGame extends BaseEntity {

    private String name;
    private String description;
    private String type;
}
```



Identyfikatory - @Id

Jednym z głównych wymagań relacyjnych baz danych **jest posiadanie unikalnych identyfikatorów**. W celu zdeklarowania kolumny, która reprezentuje klucz główny, **należy skorzystać z adnotacji @Id**. Klucz główny w Javie musi **być typem prymitywnym bądź też typem opakowującym, np.: Integer, Long, String, Date, BigInteger czy BigDecimal**.

```
@Entity  
@Table(name = "movies")  
public class Movie {  
    @Id  
    private Long id;  
  
    private String name;  
  
    @Column(name = "movie_title")  
    private String title;  
}
```

UWAGA: W praktyce, @Id bardzo często znajduje się nad polem typu Long, UUID lub String.



Identyfikatory - @GeneratedValue

Klucze główne w bazie **danych mogą być naturalne lub automatycznie generowane podczas zapisywania wiersza w bazie**. W celu oznaczenia kolumny reprezentującej identyfikator, aby był on generowany automatycznie musimy dodatkowo wykorzystać adnotację **@GeneratedValue**.

Adnotacja ma możliwość wyboru sposobu generowania takiego identyfikatora. Te sposoby **są reprezentowane przez wartości enuma GenerationType**:

- **TABLE**
- **SEQUENCE**
- **IDENTITY**



Identyfikatory - @GeneratedValue

Opis i szczegóły poszczególnych sposobów generowania są poza zakresem szkolenia. **Używając @GeneratedValue wykorzystujmy strategię GenerationType.AUTO, aby sposób generowania był automatycznie dopasowany do możliwości typu wykorzystywanej bazy danych**, np.:

```
import javax.persistence.*;

@Entity
@Table(name = "movies")
public class Movie {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;

    @Column(name = "movie_title")
    private String title;
}
```



Identityfikatory - @EmbeddedId

Tabele mogą również posiadać klucze, **które składają się z wielu kolumn**. W celu zdefiniowania takiego klucza musimy wykonać następujące kroki:

- zdefiniować **klasę, która reprezentuje klucz złożony**. Klasa ta musi implementować interfejs **Serializable**
- oznaczyć klasę reprezentującą klucz złożony adnotacją **@Embeddable**
- na encji, na której chcemy wykorzystać taki klucz, **definiujemy pole klucza i umieszczamy nad nim adnotację @EmbeddedId**.

UWAGA: Klasę, która jest reprezentacją wielu kolumn w encji, a która nie jest kluczem encji, możemy osadzić w encji w podobny sposób jak klucz złożony. W takim przypadku zamiast adnotacji @EmbeddedId, korzystamy z adnotacji **@Embedded**.



Identityfikatory - @EmbeddedId

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Embeddable;

@Embeddable
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Person {
    private String firstName;
    private String lastName;
}
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Embeddable;
import java.io.Serializable;

@Embeddable
@Data
@NoArgsConstructor
@AllArgsConstructor
public class TeacherId implements Serializable {
    private int documentId;
    private String license;
}
```



Identityfikatory - @EmbeddedId

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Embedded;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Teacher {

    @EmbeddedId // klucz złożony
    private TeacherId teacherId;

    @Embedded // osadzenie kolumn firstName i lastName w encji
    private Person person;
}
```



Identyfikatory - @IdClass

Innym sposobem definiowania kluczy złożonych z wielu kolumn jest adnotacja **@IdClass**, jednak jest on **nieco mniej intuicyjny niż wykorzystanie @EmbeddedId**.

Klasa reprezentująca klucz wygląda dokładnie tak samo, jak w przypadku wykorzystania @EmbeddedId.



Identyfikatory - @IdClass

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Embeddable;
import java.io.Serializable;

@Embeddable
@Data
@NoArgsConstructor
@AllArgsConstructor
public class TeacherId implements Serializable {
    private int documentId;
    private String license;
}
```



Identyfikatory - @IdClass

Różnica pojawia się w encji, w której definiujemy klucz złożony.

Adnotacja **@IdClass** musi znajdować się **nad definicją klasy i musi wskazywać na obiekt przechowujący informację o kolumnach składających się na klucz główny**. Kolumny te trzeba dodatkowo zdeklarować na poziomie encji bazodanowej wraz z oznaczeniem ich jako **@Id**. Rozwiążanie takie może sprawdzić się w przypadku, gdy jedna z kolumn jest relacją.



Identyfikatory - @IdClass

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@IdClass(TeacherId.class)
public class Teacher {

    @Id
    private int documentId;

    @Id
    private String license;

    @Embedded
    private Person person;
}
```



Modelowanie relacji

Za pomocą frameworka Hibernate możemy wskazać, **jak wyglądają relacje pomiędzy tabelami w relacyjnej bazie danych**. W standardowym relacyjnym modelu tabele łączone są za pomocą kluczy obcych. **W podejściu obiektowym należy nawiązać relacje pomiędzy całymi obiektami**. Hibernate pozwala zdefiniować relacje:

- jeden-do-jeden za pomocą adnotacji **@OneToOne**
- jeden-do-wielu za pomocą adnotacji **@OneToMany** i **@ManyToOne**
- wiele-do-wielu za pomocą adnotacji **@ManyToMany**



Modelowanie relacji

Wiązane relacje mogą być:

- **dwustronne (bidirectional)** - kiedy **obydwie wiązane table na poziomie klas mają referencje do siebie** (tzn. klasa A wskazuje na klasę B i B wskazuje na A)
- **jednostronne (unidirectional)** - **tylko jedna ze stron relacji zawiera referencje do tabeli reprezentowanej przez inną klasę** (tzn. klasa A wskazuje na klasę B, ale klasa B nie wskazuje na klasę A).



@OneToOne – relacja jednostronna

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity(name = "student_books")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class StudentBook {
    @Id
    private Integer id;

    private String number;
}
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import java.util.Date;

@Entity(name = "students")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Student {

    @Id
    private Integer id;

    private String firstName;

    private String lastName;

    private Date birthDate;

    @OneToOne
    private StudentBook studentBook;
}
```



@OneToOne – relacja dwustronna

Chcąc zmodyfikować relację z poprzedniego przykładu i dodać dwustronność, musimy wykorzystać adnotację **@OneToOne również na klasie StudentBook**.



@OneToOne – relacja dwustronna

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import java.util.Date;

@Entity(name = "students")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Student {

    @Id
    private Integer id;

    private String firstName;

    private String lastName;

    private Date birthDate;

    @OneToOne
    private StudentBook studentBook;
}
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity(name = "student_books")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class StudentBook {

    @Id
    private Integer id;

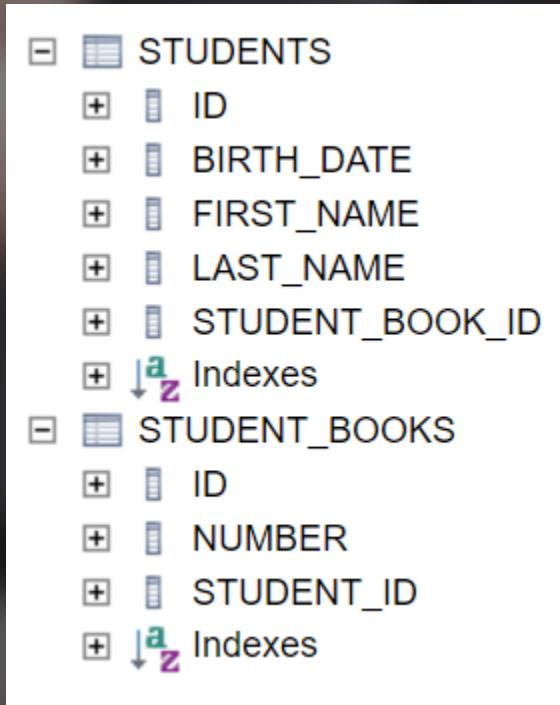
    private String number;

    @OneToOne
    private Student student;
}
```



@OneToOne – relacja dwustronna

W takim przypadku również tabela student_books będzie zawierała referencję do wartości obiektu Student, tzn.:





@JoinColumn

Domyślnie Hibernate oczekuje, że kolumna wskazująca na "drugą stronę" relacji ma nazwę `nazwa_encji_pole_identyfikatora`. Nazwę tę możemy zmienić za pomocą adnotacji `@JoinColumn`.



@JoinColumn

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import java.util.Date;

@Entity(name = "students")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Student {

    @Id
    private Integer id;

    private String firstName;

    private String lastName;

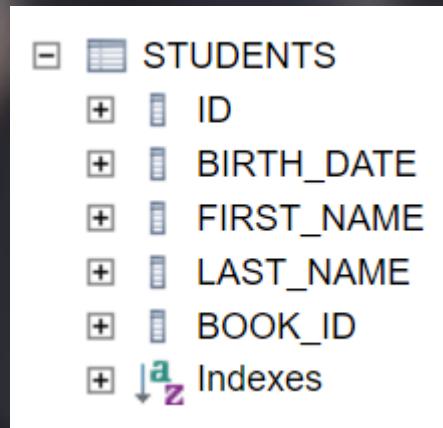
    private Date birthDate;

    @OneToOne
    @JoinColumn(name = "book_id")
    private StudentBook studentBook;
}
```



@JoinColumn

Dzięki temu, na poziomie bazy danych **możemy użyć dowolnej nazwy kolumny, tzn. w tym przypadku book_id**:





mappedBy

Każda z omawianych asocjacji **umożliwia określenie właściwości mappedBy**. Parametr ten wskazuje na **nazwę pola w klasie, która jest właścicielem relacji** (np. posiada referencję do drugiej tabeli za pomocą klucza obcego).

Kolejny przykład modyfikuje relacje pomiędzy encjami zdefiniowanymi w poprzednich przykładach - Student i StudentBook. Tym razem to encja Student jest właścicielem relacji. Zauważmy, że:

- **klasa StudentBook wskazuje właściciela relacji za pomocą nazwy pola**
- właściciel jest wskazany w atrybucie **mappedBy** w adnotacji **@OneToOne**
- na poziomie bazy danych, **tabela StudentBook nie powinna w takim przypadku zawierać identyfikatora encji Student**



mappedBy

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import java.util.Date;

@Entity(name = "students")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Student {

    @Id
    private Integer id;

    private String firstName;

    private String lastName;

    private Date birthDate;

    @OneToOne
    @JoinColumn(name = "book_id")
    private StudentBook studentBook;
}
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity(name = "student_books")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class StudentBook {

    @Id
    private Integer id;

    private String number;

    @OneToOne(mappedBy = "studentBook")
    private Student student;
}
```



Relacje 1-do-wielu/wiele-do-1

Relację jeden-do-wielu te **можемy definiować za pomocą adnotacji @OneToMany**. Adnotację taką wykorzystujemy na polu klasy reprezentującą kolekcję zależnej encji.

Relację jeden-do-wielu możemy stworzyć na dwa sposoby:

- podobnie jak w relacji wiele-do-wielu, za pomocą tabeli łączącej
- **za pomocą dodania dodatkowej kolumny wskazującej na właściciela relacji.**

W praktyce rzadko wykorzystujemy tabelę łączącą do zdefiniowania takiej relacji.



Relacje 1-do-wielu/wiele-do-1 – bez tabeli łączącej

Najczęściej wygodniejszym sposobem **definiowania relacji jeden-do-wielu jest wykorzystanie dodatkowej kolumny w tabeli**, która reprezentuje część relacji "wielu". Efekt ten możemy uzyskać poprzez dodanie adnotacji **@JoinColumn** obok adnotacji **@OneToOne**

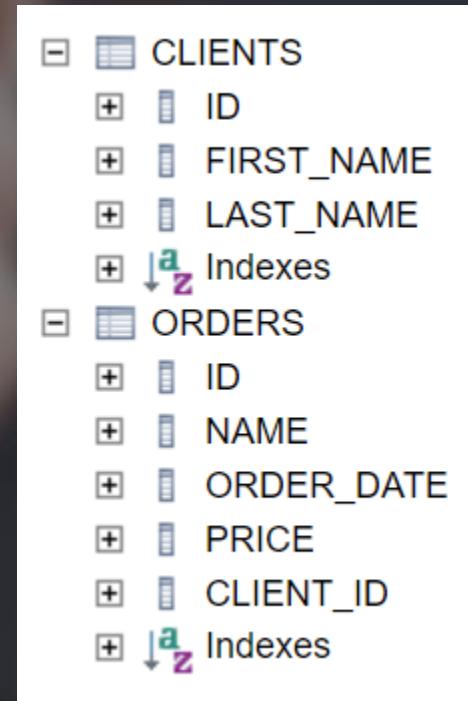


Relacje 1-do-wielu/wiele-do-1 – bez tabeli łączącej

```
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Entity(name = "clients")  
public class Client {  
    @Id  
    private Integer id;  
  
    private String firstName;  
    private String lastName;  
  
    @OneToMany  
    @JoinColumn(name = "client_id")  
    private List<Order> orders;  
}
```



Relacje 1-do-wielu/wiele-do-1 – bez tabeli łączącej





Relacje 1-do-wielu/wiele-do-1 – relacja dwustronna

Opisywana relacja z poprzednich przykładów może być również dwustronna. **Efekt ten uzyskujemy poprzez dodanie adnotacji `@ManyToOne`.**

Wykorzystując adnotacje `@OneToMany` i `@ManyToOne`, bez `@JoinColumn`, Hibernate oczekuje tabeli łączącej te dwie encje oraz dodatkowej kolumny wskazującej na ojca relacji.



Relacje 1-do-wielu/wiele-do-1 – relacja dwustronna

Tak jak w przypadku wykorzystania relacji **jeden-do-jednego**, **istnieje także możliwość wskazania właściciela relacji, za pomocą pola mappedBy**. Wartość atrybutu wskazuje na nazwę pola po drugiej stronie relacji.



Relacje 1-do-wielu/wiele-do-1 – relacja dwustronna

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import java.util.List;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity(name = "clients")
public class Client {
    @Id
    private Integer id;

    private String firstName;
    private String lastName;

    @OneToMany(mappedBy = "client")
    private List<Order> orders;
}
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import java.util.Date;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity(name = "orders")
public class Order {

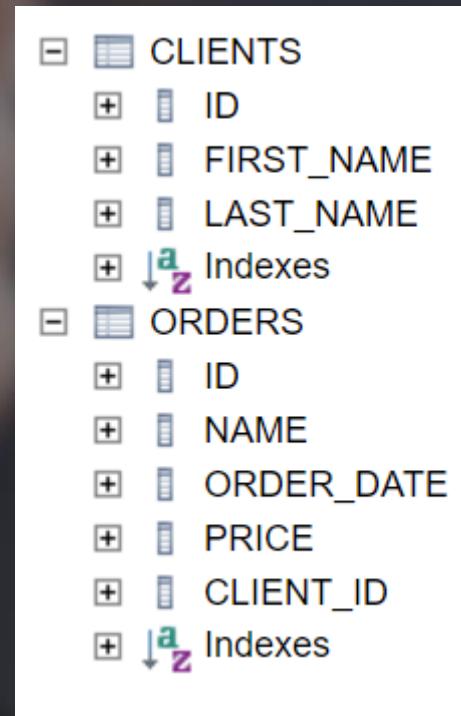
    @Id
    private Integer id;

    private float price;
    private String name;
    private Date orderDate;

    @ManyToOne
    private Client client;
}
```



Relacje 1-do-wielu/wiele-do-1 – relacja dwustronna





@ManyToMany

Relacyjne bazy danych dają również możliwość tworzenia relacji **wiele-do-wielu**. Za pomocą Hibernate efekt taki możemy uzyskać wykorzystując adnotację **@ManyToMany**. Relacja taka:

- **zawsze musi posiadać tabelę łączącą**
- **może być jednostronna lub dwustronna**
- **może wskazywać właściciela relacji za pomocą atrybutu mappedBy**

Kolejny przykład wykorzystuje encje Orders i Client, ale tym razem jedno zamówienie może być przypisane do wielu klientów. Ponadto jeden klient może mieć wiele zamówień. W przykładzie została wykorzystana relacja jednostronna.



@ManyToMany

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;
import java.util.Date;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity(name = "orders")
public class Order {

    @Id
    private Integer id;

    private float price;
    private String name;
    private Date orderDate;
}
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import java.util.List;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity(name = "clients")
public class Client {

    @Id
    private Integer id;

    private String firstName;
    private String lastName;

    @ManyToMany
    private List<Order> orders;
}
```



@ManyToMany

-	CLIENTS
+	ID
+	FIRST_NAME
+	LAST_NAME
+	↓ a z Indexes
-	CLIENTS_ORDERS
+	CLIENTS_ID
+	ORDERS_ID
+	↓ a z Indexes
-	ORDERS
+	ID
+	NAME
+	ORDER_DATE
+	PRICE
+	↓ a z Indexes



Dziękuję