



Interactive Web Applications



1

Object-Relational Mapping

(OneToOne, OneToMany, ManyToOne, ManyToMany)

Modern database technologies have advanced to keep up with the increasing volume of data:

- **relational databases** - continue to be widely used for transactional applications and data warehousing;
- **NoSQL databases** - gained popularity for their ability to handle large volumes of unstructured data and provide high scalability and performance;
- **cloud databases** - offer scalable and flexible database solutions with minimal infrastructure management.

Relational databases are a core component of any modern transactional application. The relational model is composed of tables (data organized in rows and columns) that have at least one unique key that identifies each row. Each table represents an entity. Each row in a table represents a single record, and the columns represent the different attributes or characteristics of that record.



Relational databases offer many advantages over other types of databases. Some of the main advantages include:

- data consistency
- data integrity
- data security
- scalability
- query flexibility

The **ACID** properties of the transaction: **Atomicity**, **Consistency**, **Isolation**, and **Durability** - are a set of principles that are widely used in the design of **transactional systems**. They are particularly relevant in relational database management systems to ensure data consistency, reliability, and integrity.

Atomicity – all operations in a transaction (to read, write, update or delete data) are treated as a single unit. Either all of them are executed, or none of them is executed. This property prevents data loss and corruption from occurring.

Consistency – ensures that transactions only make changes to tables in predefined, predictable ways. It ensures that errors in the data do not create unintended consequences for the integrity of the table and the database.

Isolation – when multiple users are reading and/or writing from the same table all at once, isolation of their transactions ensures that the concurrent transactions do not interfere with or affect one another. Each request can occur as though they were occurring one by one, even though they are actually occurring simultaneously.

Durability – ensures that changes to the data made by successfully executed transactions will be saved, even in the event of system failure.

Relationships between tables in the database are crucial.

Relationships between objects:

- **1-1 (@OneToOne)**
- **1-* (@OneToMany)**
- ***-1 (@ManyToOne)**
- ***-* (@ManyToMany)**

Each type of a relationship can be **one-way** or **two-way**.
A relationship has an owning side and an inverse side.

Typical properties of a relationships (annotation parameters):

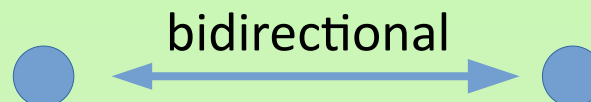
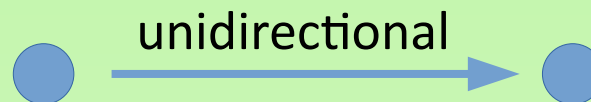
- **cascade** – what operations can be cascaded on the relationship (all, write to DB, delete, update)
- **fetch** – LAZY | EAGER
- **mappedBy** – on the "inverse" side, specifying which attribute of the "owning" side points to the given object. Relationships are always mapped on the owner side.

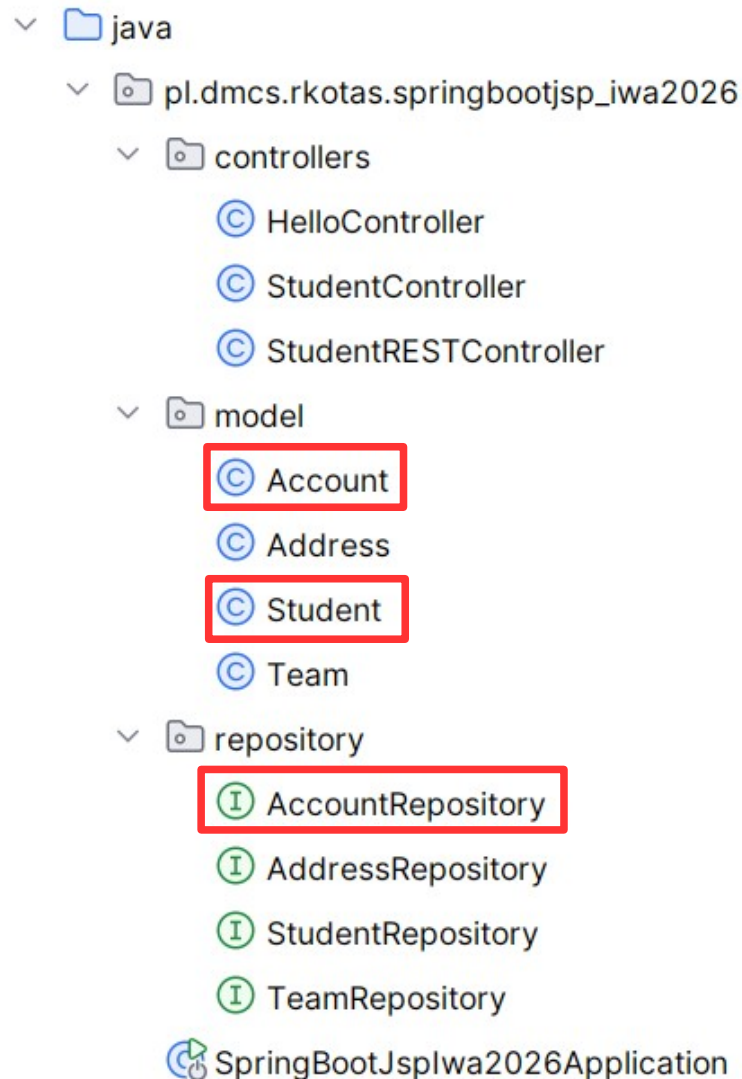
One-to-one

It is a relationship where a record in one entity (table) is associated with exactly one record in another entity (table).

Real-life examples of one-to-one relationships:

- Country – capital city
- Email – user account
- Person – set of fingerprints
- User profile – set of user settings

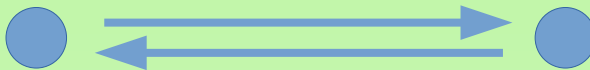




Object-Relational Mapping - @OneToOne

* two unidirectional relations

```
Student.java x
DDL [a] [m] [f] [Q] [G] [↕]
6 public class Student {
16 @OneToOne(cascade = CascadeType.ALL)
17 private Account account;
```



```
Account.java x
DDL [a] [m] [f] [Q] [G] [↕]
8 @Entity
9 public class Account {
10
11 @Id
12 @GeneratedValue
13 private long id;
14 private String accountName;
15
16 @OneToOne
17 private Student student;
```

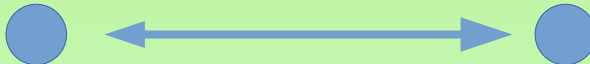
	id [PK] bigint	email character varying (255)	firstname character varying (255)	lastname character varying (255)	telephone character varying (255)	account_id bigint
1	152	rkotas2@dmcs.pl	Rafal	Kotas	456456456	1

	id [PK] bigint	account_name character varying (255)	student_id bigint
1	1	dmcsAccount	[null]

Object-Relational Mapping - @OneToOne

* one bidirectional relation

```
Student.java x
DDL  [a] [m] [k] [Q] [G] [↕]
6      public class Student {
16      @OneToOne(cascade = CascadeType.ALL)
17      private Account account;
```



```
Account.java x
DDL  [a] [m] [k] [Q] [G] [↕]
8      @Entity
9      public class Account {
10
11          @Id
12          @GeneratedValue
13          private long id;
14          private String accountName;
15
16          @OneToOne(mappedBy = "account")
17          private Student student;
```

	id [PK] bigint	email character varying (255)	firstname character varying (255)	lastname character varying (255)	telephone character varying (255)	account_id bigint
1	102	rkotas@dmcs.pl	Rafal	Kotas	456456456	102

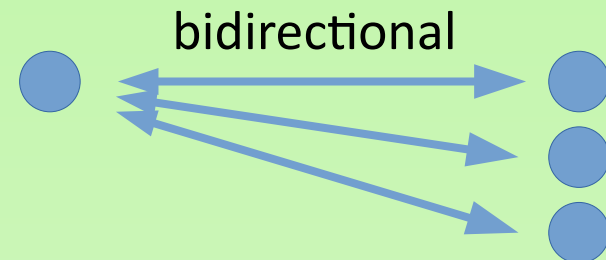
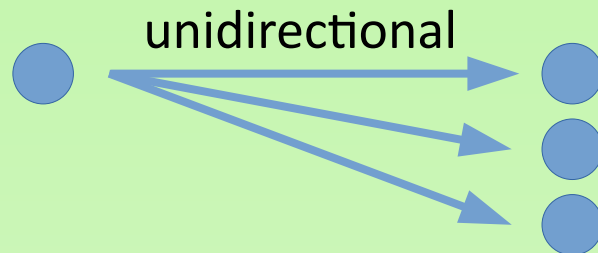
	id [PK] bigint	account_name character varying (255)
1	102	dmcsAccount

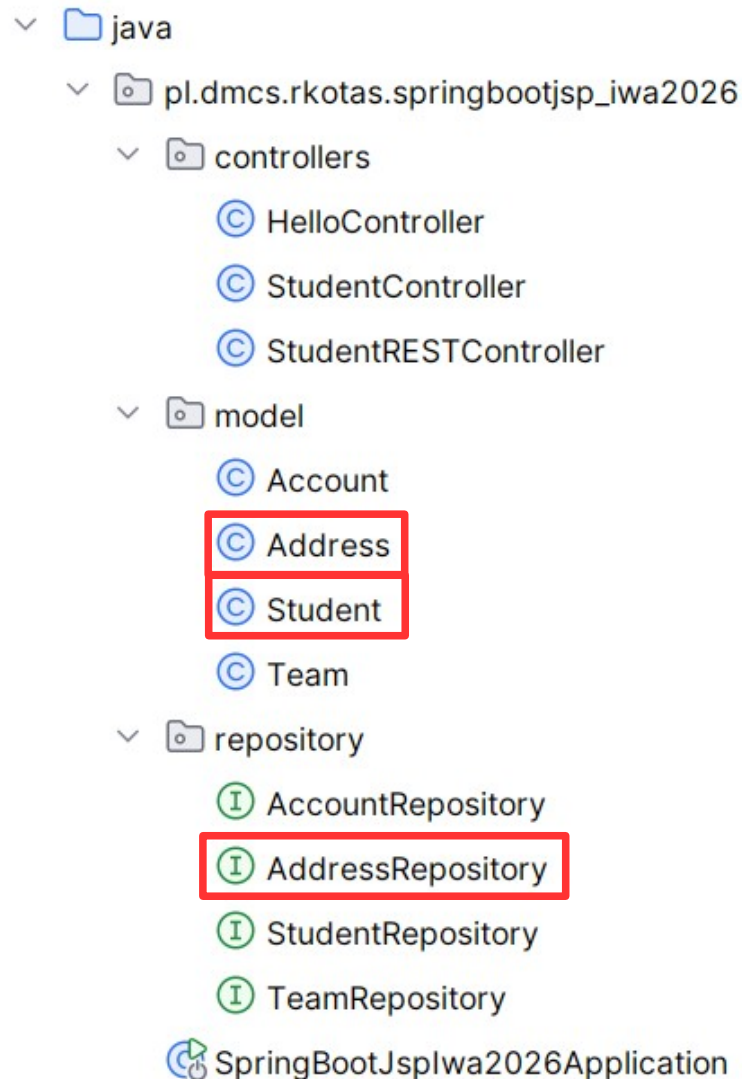
One-to-many

It is the most common type of relationship, where a record in one entity can be referenced by multiple records in another entity.

Real-life examples of one-to-many relationships:

- Music playlists – songs
- Courses – students
- Teachers – classes
- Client – orders





Object-Relational Mapping - @ManyToOne and @OneToMany

* bidirectional relation

```
Student.java x
DDL  [a] [m] [f] [Q] [Q] [Q] [Q]
6      public class Student {
19      @ManyToOne(cascade = CascadeType.MERGE)
20      private Address address;
```

```
Address.java x
DDL  [a] [m] [f] [Q] [Q] [Q] [Q]
8      public class Address {
19      @OneToMany(mappedBy = "address", fetch = FetchType.EAGER)
20      private List<Student> studentList;
```

or

Tables (5)

- > account
- > address
- > student
- > student_team_list
- > team

```
Address.java x
DDL  [a] [m] [f] [Q] [Q] [Q] [Q]
8      public class Address {
19      @OneToMany(mappedBy = "address", fetch = FetchType.EAGER)
20      @JoinTable(name = "student_address", joinColumns = @JoinColumn(name = "student_id"),
21                inverseJoinColumns = @JoinColumn(name = "address_id"))
22      private List<Student> studentList;
```

Tables (6)

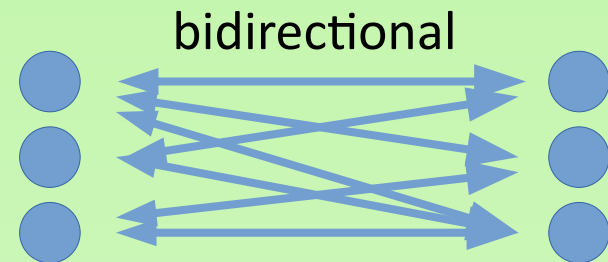
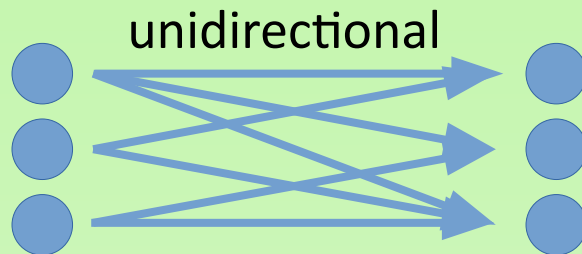
- > account
- > address
- > student
- > **student_address**
- > student_team_list
- > team

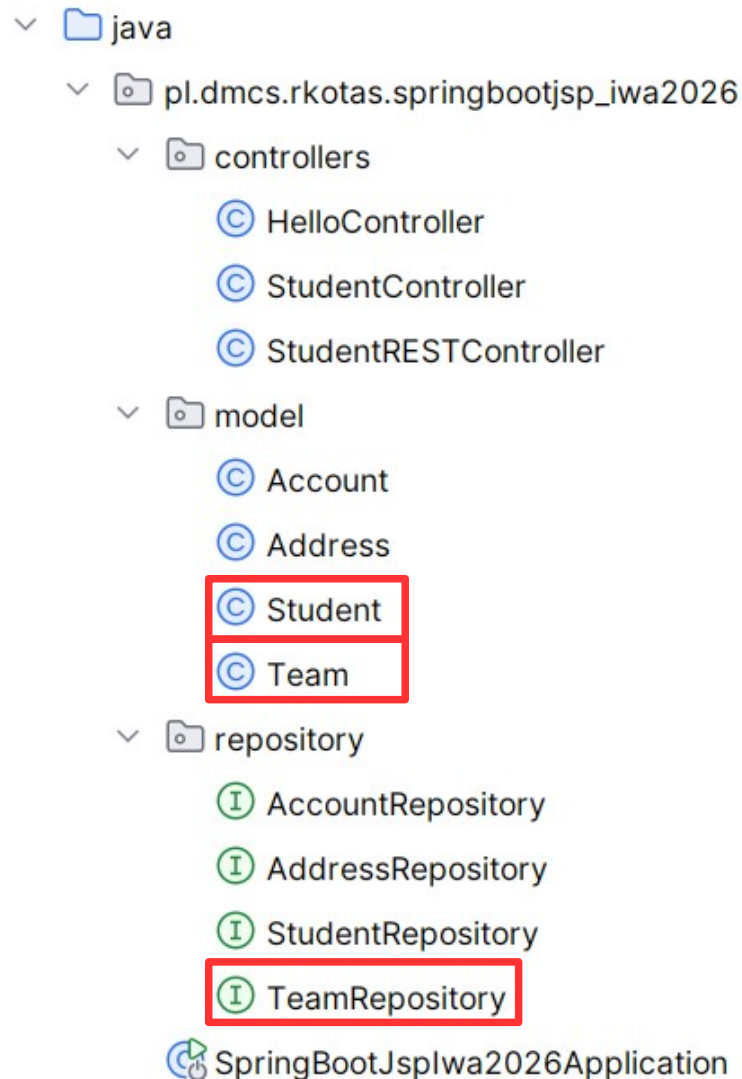
Many-to-many

It is a relationship where more than one record in a table is related to more than one record in another table.

Real-life examples of many-to-many relationships:

- Customers – products
- Users – roles (authorities)
- Students – classes





* unidirectional relation

```
Student.java x
DDL  [a] [m] [f] [d] [u]
8      public class Student {
24      @ManyToMany(cascade = CascadeType.PERSIST)
25      private List<Team> teamList;
```

```
Team.java x
DDL  [a] [m] [f] [d] [u]
7      @Entity
8      public class Team {
9
10         @Id
11         @GeneratedValue
12         private long id;
13         private String teamName;
```

Tables (5)

- account
- address
- student
- student_team_list
- team

	id [PK] bigint	email character varying (255)	firstname character varying (255)	lastname character varying (255)	telephone character varying (255)	account_id bigint	address_id bigint
1	152	rkotas@dmcs.pl	Rafal	Kotas	123123123	152	1
2	153	jnowak@dmcs.pl	Jan	Nowak	444555666	153	2

	student_id bigint	team_list_id bigint
1	152	1
2	152	2
3	153	3

	id [PK] bigint	team_name character varying (255)
1	1	teamA
2	2	teamB
3	3	teamC

Object-Relational Mapping – Controllers

© StudentRestController.java ×

```
13 @RestController
14 @RequestMapping("/students")
15 public class StudentRestController {
16
17     private StudentRepository studentRepository;
18     private AddressRepository addressRepository;
19
20     @Autowired
21     public StudentRestController(StudentRepository studentRepository, AddressRepository addressRepository) {
22         this.studentRepository = studentRepository;
23         this.addressRepository = addressRepository;
24     }
25
26     @RequestMapping(method = RequestMethod.GET, produces = "application/xml")
27     // @GetMapping
28     public List<Student> findAllStudents() { return studentRepository.findAll(); }
29
30
31
32     @RequestMapping(method = RequestMethod.POST)
33     // @PostMapping
34     public ResponseEntity<Student> addStudent(@RequestBody Student student) {
35         if (student.getAddress().getId() <= 0 )
36         {
37             addressRepository.save(student.getAddress());
38         }
39         studentRepository.save(student);
40         return new ResponseEntity<Student>(student, HttpStatus.CREATED);
41     }
```

Object-Relational Mapping – Postman tests



21

dr inż. Rafał Kotas, rkotas@dmcs.pl

POST localhost:8080/students

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▼

```
1 {
2   "firstname": "Rafal",
3   "lastname": "Kotas",
4   "email": "rkotas@dmcs.pl",
5   "telephone": "123123123",
6   "account": { "accountName": "dmcsAccount" },
7   "address": { "city": "Lodz", "street": "Piotrkowska", "number": "4", "postalCode": "99-999" },
8   "teamList": [ { "teamName": "teamA" }, { "teamName": "teamB" } ]
9 }
```

POST localhost:8080/students

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▼

```
1 {
2   "firstname": "Rafal2",
3   "lastname": "Kotas2",
4   "email": "rkotas2@dmcs.pl",
5   "telephone": "456456456",
6   "account": { "accountName": "dmcsAccount2" },
7   "address": { "id": "4" },
8   "teamList": [ { "teamName": "teamA2" } ]
9 }
```

Infinite recursion - (**StackOverflowError**) - bi-directional association/relationship

1. Improve model and avoid unnecessary bidirectional relationships.
2. @JsonIgnore
3. @JsonIgnoreProperties
4. @JsonManagedReference and @JsonBackReference
5. @JsonView

[illegible]

Object-Relational Mapping – Postman tests



24

dr inż. Rafał Kotas, rkotas@dmcs.pl

```
Address.java x
public class Address {
    @JsonManagedReference
    @OneToMany(mappedBy = "address", fetch = FetchType.EAGER)
    // @JoinTable(name="student_address", joinColumns = @JoinColumn(name="student_id"),
    //           inverseJoinColumns = @JoinColumn(name="address_id"))
    private List<Student> studentList;
}
```

```
Student.java x
public class Student {
    @JsonBackReference
    @ManyToOne(cascade = CascadeType.MERGE)
    private Address address;
}
```

```
Account.java x
@JsonIgnoreProperties(ignoreUnknown = true, value = {"student", "id"})
@Entity
public class Account {
    @Id
    @GeneratedValue
    private long id;

    private String accountName;

    // @JsonIgnore
    @OneToOne(mappedBy = "account")
    private Student student;
}
```

GET localhost:8080/students

Params Authorization Headers (7) Body Pre

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
{
  "id": 5,
  "firstname": "Rafal",
  "lastname": "Kotas",
  "email": "rkotas@dmcs.pl",
  "telephone": "123123123",
  "account": {
    "id": 6,
    "accountName": "dmcsAccount"
  },
  "teamList": [
    {
      "id": 7,
      "teamName": "teamA"
    },
    {
      "id": 8,
      "teamName": "teamB"
    }
  ]
}
```


1. Implement relationships from the above example
2. Add endpoints for all new entities (get, post, delete, put)
3. Check and test different configurations (modify controller methods if necessary):
 - Unidirectional and bidirectional relations
 - Cascading types
 - Infinite recursions
4. *Homework: Do the reading on: „database normalization” and „normal forms”.*



THE END

WIKAMP → IWA_4.zip

