Spring Data - I

Teaching Faculty: Dr. Muhyieddin Al-Tarawneh

Prepared by Muhyieddin **AL-TARAWNEH**, Umur **INAN**

JDBC

- It stands for Java Database Connectivity.
- It provides a set of Java API for accessing the relational databases from Java program.
- It provides a flexible architecture to write a database independent application that can run on different platforms and interact with different DBMS without any modification.

JDBC use cases

- Making a connection to a database.
- Creating SQL statements.
- Executing SQL queries in the database.
- Viewing & Modifying the resulting records.

JDBC

Pros

- Clean and simple SQL processing
- Good performance with large data
- Very good for small applications
- Simple syntax so easy to learn

Cons

- Complex if it is used in large projects
- Large programming overhead
- No encapsulation
- Query is DBMS specific

JPA – JAVA PERSISTENCE API

- It is a Java specification for accessing, persisting, and managing data between Java objects / classes and a relational database.

- It is now considered the standard industry approach for Object to Relational Mapping (ORM) in the Java Industry.

- JPA itself is just a specification, not a product; it cannot perform persistence or anything else by itself. JPA is just a set of interfaces and requires an implementation.

JPA PROVIDERS

- Hibernate
- Eclipselink
- Toplink

ORM – Object relational mapping

- It Acts as a 'Gateway' between OO Domain && Relational Database.
- It Maps Object to Relational Model & Vice Versa.
- ORM tools essentially present a relational database from an object-oriented viewpoint.
- The ORM is not enhancing the Domain Model, it is simply a tool to overcome the O/R differences & to hide SQL.

ORM advantages

- Business code access objects rather than DB tables.
- Hides details of SQL queries from OO logic.
- Based on JDBC 'under the hood.'
- No need to deal with the database implementation.
- Entities based on business concepts rather than database structure.

Basic ORM features

- Mapping Classes To Tables
- Out Of The Box CRUD Functionality
- Hydrating Entities
- Executing Custom “OO” Queries
- Cache management
- Concurrency support
- Transaction management

Defining entity class

- @Entity
  - It identifies a class as an entity class.

- @Table
  - By default, each entity class maps a database table with the same name in the default schema of your database.
  - Customize this mapping using the name, schema, and catalog attributes of the @Table annotation.

Defining entity class

- @Column
  - It is an optional annotation that enables to customize the mapping between the entity attribute and the database column.
  - use the name attribute to specify the name of the database column
  - The length attribute, which defines the length of String-valued database column.

Defining entity class

- @Id
  - JPA and Hibernate require to specify at least one primary key attribute for each entity.

- @GeneratedValue
  - use a database sequence by setting the strategy attribute to GenerationType.SEQUENCE
  - use an auto-incremented database column to generate your primary key values by setting strategy to GenerationType.IDENTITY.

Defining entity class

- @Id
  - marks a field in a model class as the primary key.
  - JPA and Hibernate require to specify at least one primary key attribute for each entity.
- @GeneratedValue
  - use a database sequence by setting the strategy attribute to GenerationType.SEQUENCE
  - use an auto-incremented database column to generate your primary key values by setting strategy to GenerationType.IDENTITY.

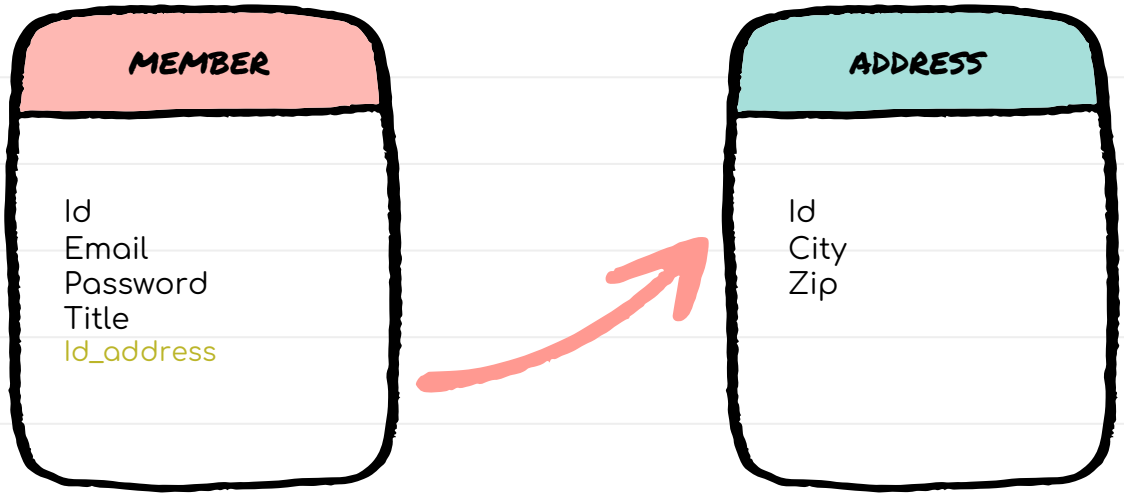
One-to-one unidirectional

Foreign Key 'id_address' will be created on Member table

```java
@Entity
public class Member {
    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    private Long id;
    private String email;
    private String password;
    private String title;

    @OneToOne
    @JoinColumn(name = "id_address") // OPTIONAL
    private Address address;
}
```

```java
@Entity
public class Address {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String zip;
    private String city;

}
```

One-to-one unidirectional



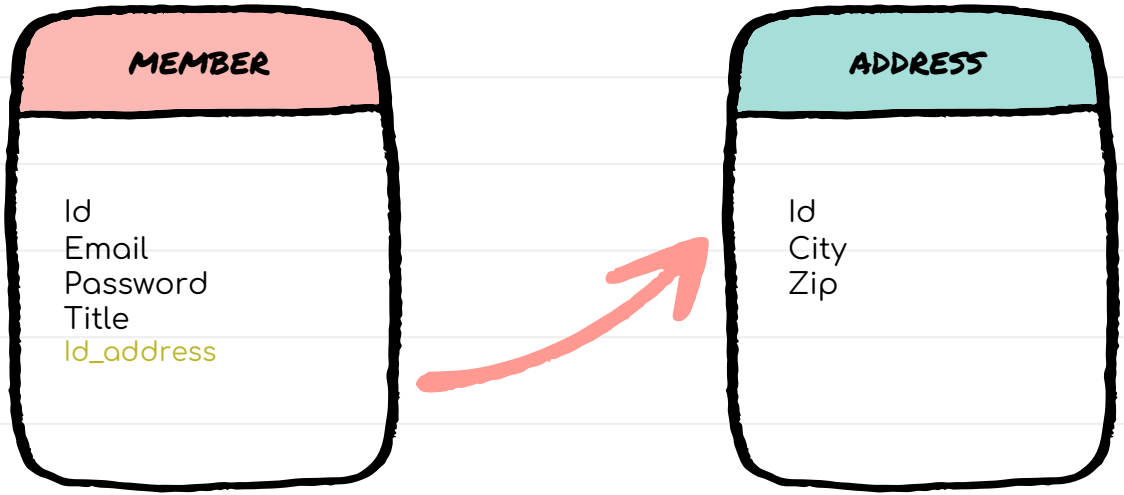
One-to-one bidirectional

```java
@Entity
public class Member {
    @Id
    @GeneratedValue(strategy
   = GenerationType.IDENTITY)
    private Long id;
    private String email;
    private String password;
    private String title;

    @OneToOne
    @JoinColumn(name = "id_address") // OPTIONAL
    private Address address;
}
```

```java
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy
   = GenerationType.IDENTITY)
    private Long id;

    private String zip;
    private String city;

    @OneToOne(mappedBy = "address")
    private Member member;
}
```

Foreign Key 'id_address' will be created on Member table.

One-to-one bidirectional



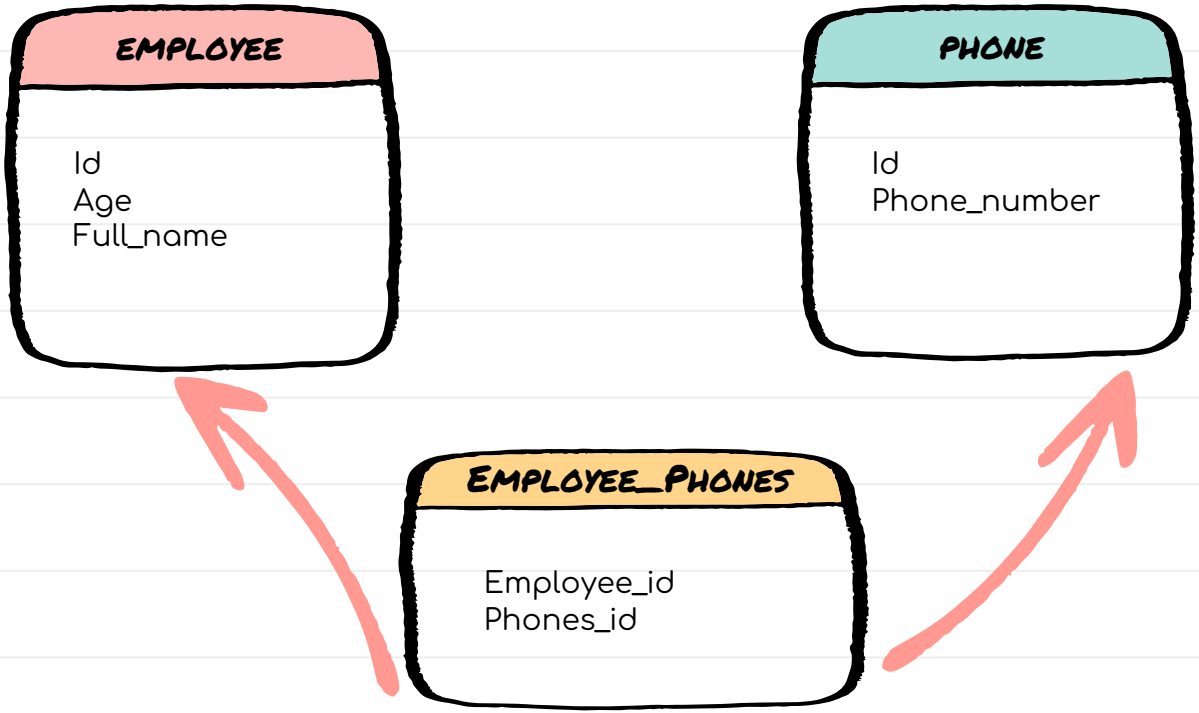
One-to-many uni-directional - join table

```java
@Entity
public class Employee {

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

    private String fullName;
    private int age;

    @OneToMany
    private List<Phone> phones;
}
```

```java
@Entity
public class Phone {

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

    private String phoneNumber;
}
```

Will create a Join Table.

One-to-many uni-directional - join table



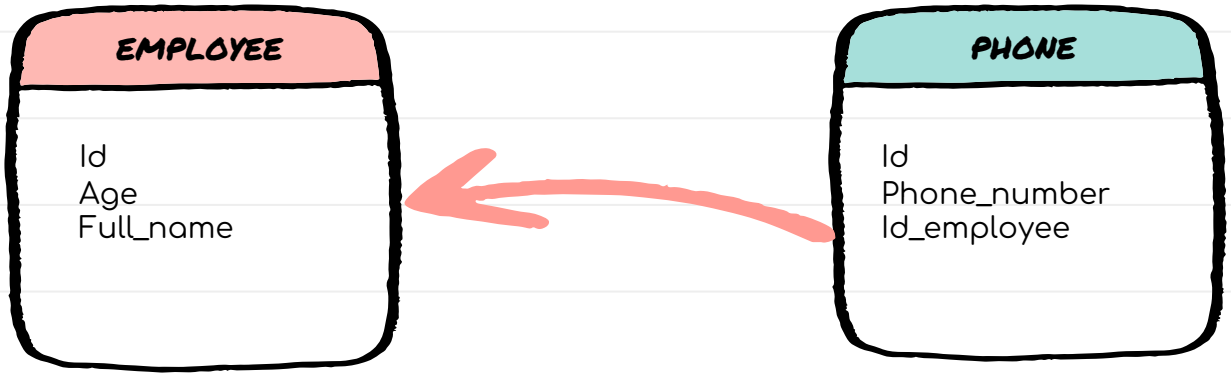
One-to-many uni-directional - join column

```java
@Entity
public class Employee {

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

    private String fullName;
    private int age;

    @OneToMany
    @JoinColumn(name = "id_employee")
    private List<Phone> phones;
}
```

```java
@Entity
public class Phone {

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

    private String phoneNumber;
}
```

Foreign Key 'id_employee will be created on Phone table.

One-to-many uni-directional - Join column



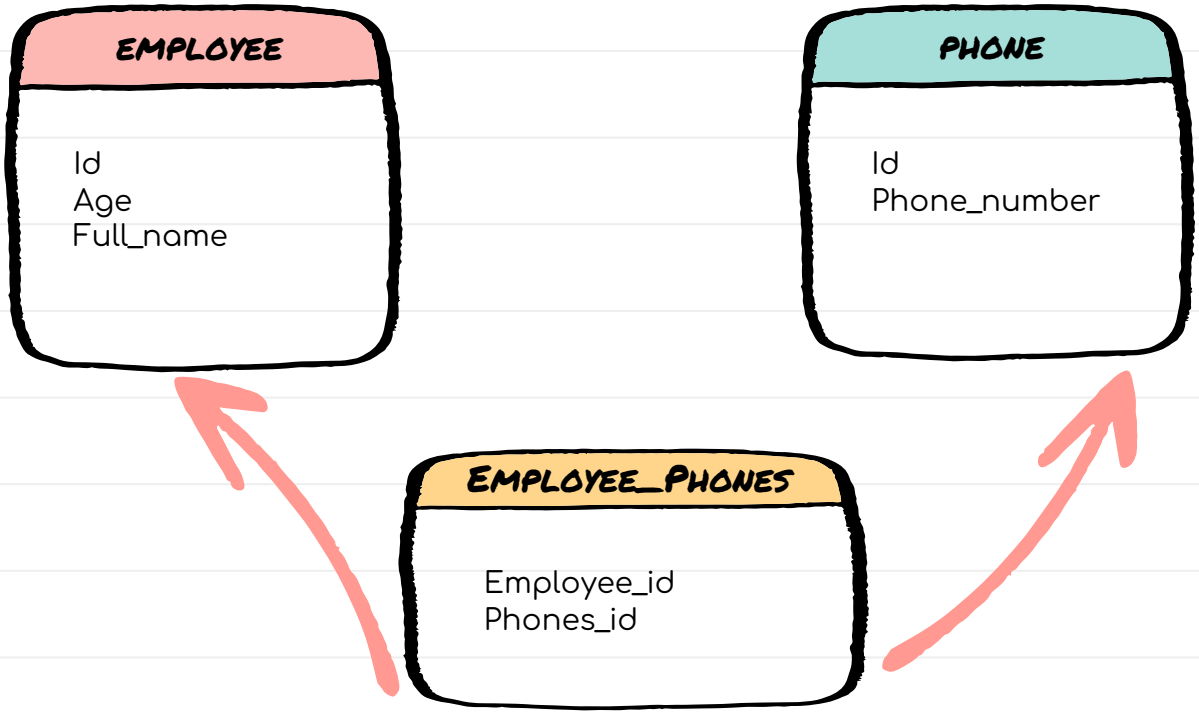
One-to-many bidirectional - join table

```java
@Entity
public class Employee {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   private String fullName;
   private int age;

   @OneToMany
   @JoinTable  // OPTIONAL
   private List<Phone> phones;
}
```

```java
@Entity
public class Phone {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   private String phoneNumber;

   @ManyToOne
   private Employee employee;
}
```

Employee_phones table will be created.

One-to-many bidirectional - join table



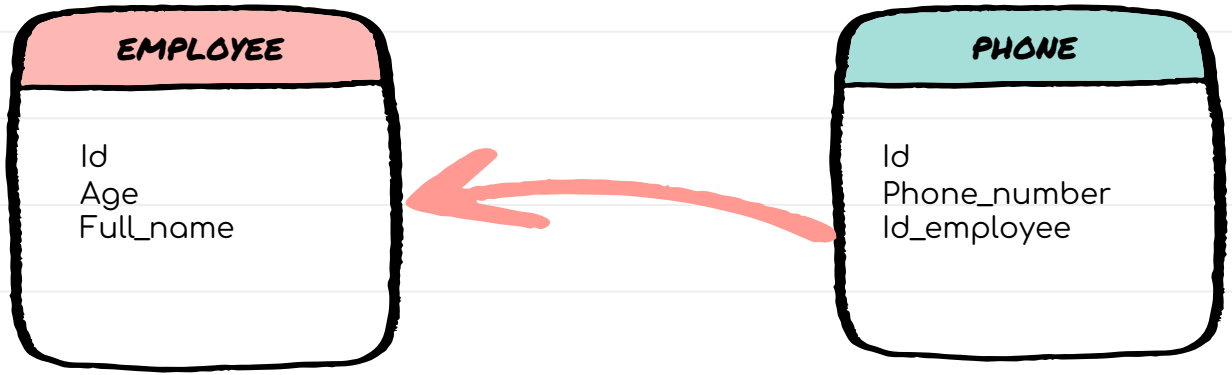
One-to-many bidirectional - join column

```java
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy
   = GenerationType.IDENTITY)
    private Long id;


    private String fullName;
    private int age;


    @OneToMany(mappedBy = "employee")
    private List<Phone> phones;
}
```

```java
@Entity
public class Phone {
    @Id
    @GeneratedValue(strategy
  = GenerationType.IDENTITY)
    private Long id;


    private String phoneNumber;


    @ManyToOne
    @JoinColumn // OPTIONAL
    private Employee employee;
}
```

Foreign Key
id_employee will be created on Phone table.

One-to-many bidirectional - join column

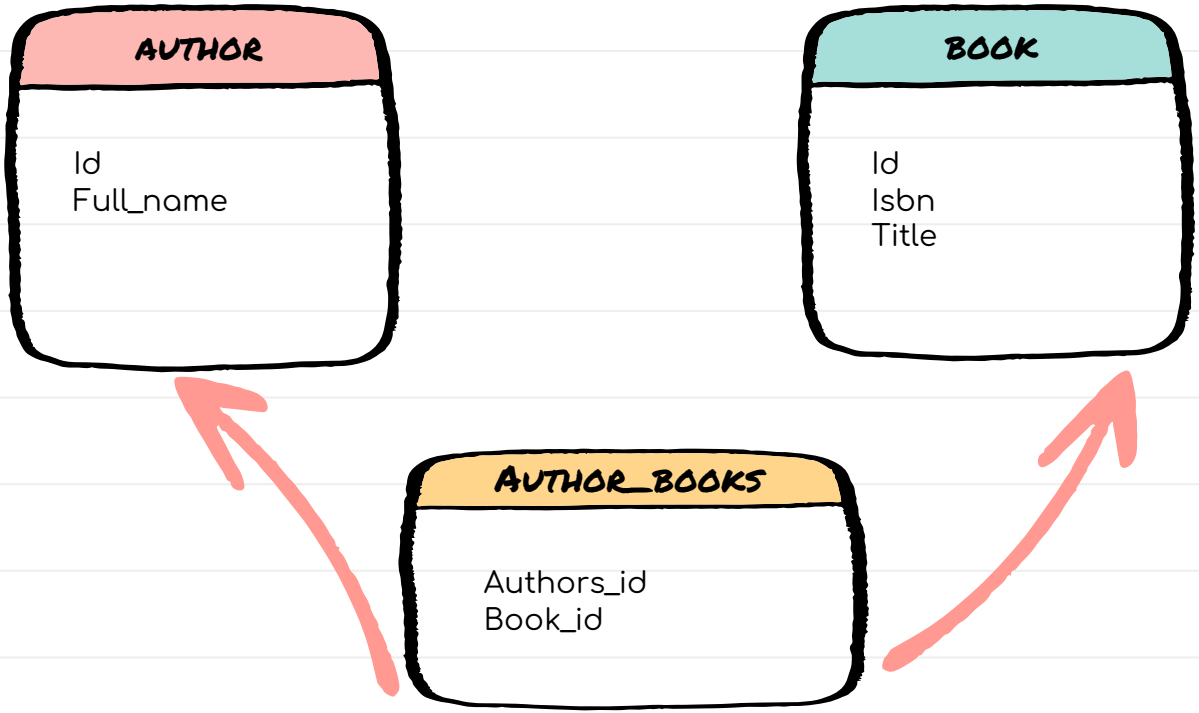


Many-to-Many

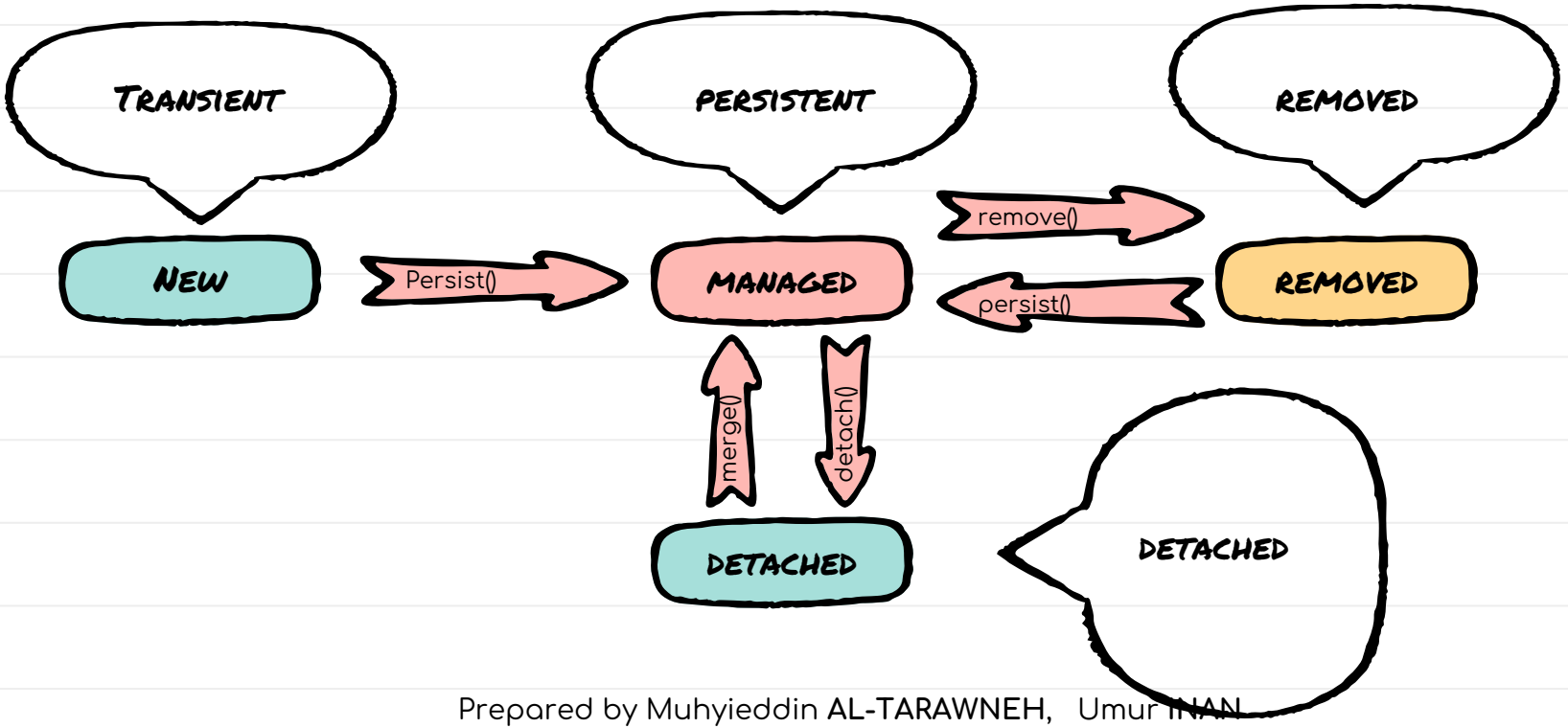
```java
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    private Long id;
    private String fullName;

    @ManyToMany
    private List<Book> books;
}
```

```java
@Entity
public class Book {

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

    private String title;
    private String isbn;

    @ManyToMany(mappedBy = "books")
    private List<Author> authors;
}
```

Many-to-many


AUTHOR
Id
Full_name
BOOK
Id
Isbn
Title
AUTHOR_BOOKS
Authors_id
Book_id

ORM ENTITY LIFECYCLE



ORM entity lifecycle

- Transient
  - It has just been instantiated using the new operator.
  - Not associated with a Persistence Context.
  - No persistent representation in the database.
- Persistent
  - Representation in the database.
  - Has been saved or loaded in Persistence Context.
  - Changes made to an object are synchronized with the database when the unit of work completes.

ORM entity lifecycle

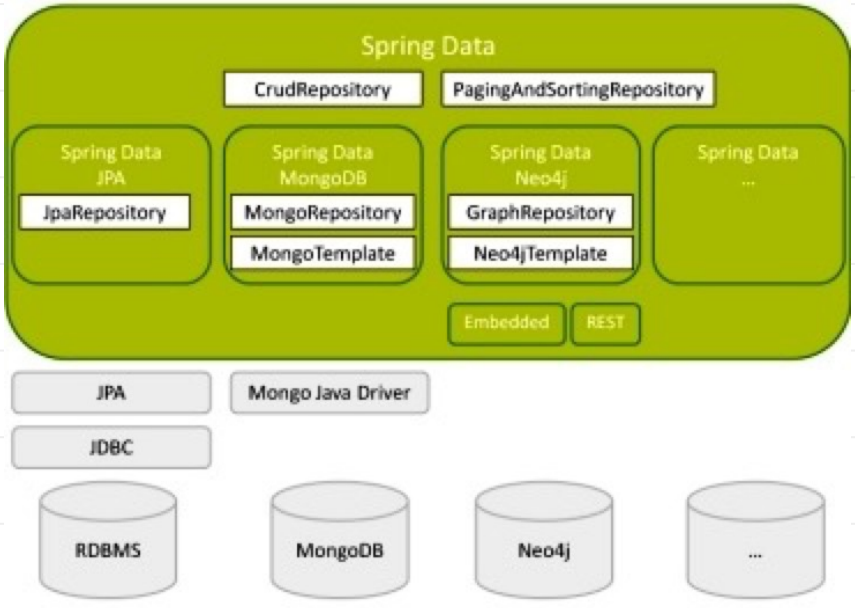
- Detached
  - Object was persistent, but Persistence Context has been closed.

- Removed
  - An object is deleted from the database when the unit of work completes.

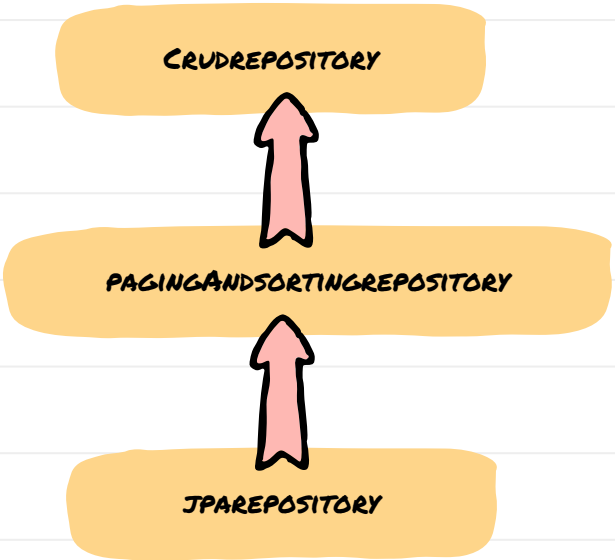
Spring data repositories

- Spring Data repository abstraction.
- Significantly reduce the amount of boilerplate code required to implement data access layers.
- Performs function of a Base Class DAO.

Spring data


Spring Data
CrudRepository
PagingAndSortingRepository
Spring Data JPA
JpaRepository
Spring Data MongoDB
MongoRepository
MongoTemplate
Spring Data Neo4j
GraphRepository
Neo4jTemplate
Spring Data ...
Embedded
REST
JPA
Mongo Java Driver
JDBC
RDBMS
MongoDB
Neo4j
...

Spring data repositories



Crud repository

- Provides CRUD functions
  - deleteById(ID id)
  - existsById(ID id)
  - findAll()
  - findAllById(Iterable<ID> ids)
  - findById(ID id)
  - save(S entity)
  - saveAll(Iterable<S> entities)

Crud repository

- Provides CRUD functions
  - count()
  - delete(T entity)
  - deleteAll()
  - deleteAll(Iterable<? extends T> entities)
  - deleteAllById(Iterable<? extends ID> ids)

Paging and sorting repository

- Provides methods to do pagination and sorting records.
  - findAll(Pageable pageable)
  - findAll(Sort sort)

Jpa repository

- provides methods such as flushing the persistence context and delete record in a batch.

Derived query methods – naming convention

- Just by looking at the corresponding method name in the code, Spring Data JPA can determine what the query should be.

- Spring Data JPA supports
  - find
  - read
  - query
  - count
  - get

EXAMPLES

- List<T> findByAgeLessThan(Integer age)
- List<T> findByNameIsNot(String name);
- List<T> findByActiveTrue();
- List<T> findByNameStartingWith(String prefix);

EXAMPLES

- List<T> findByNameEndingWith(String suffix);
- List<T> findByNameContaining(String infix);
- List<T> findByNameOrBirthDateAndActive(String name, ZonedDateTime birthDate, Boolean active);
- List<User> findByNameOrderByNameAsc(String name);

JPQL

- Java Persistence Query Language (JPQL) is an object model focused query language similar in nature to SQL.
- JPQL understands notions like inheritance, polymorphism and association.
- JPQL is a heavily-inspired-by a subset of HQL. A JPQL query is always a valid HQL query, the reverse is not true, however.
- Prevents SQL injection.

JPQL SYNTAX

- CLAUSES:
  - SELECT, FROM, WHERE, GROUP BY, HAVING and ORDER BY
- OPERATORS:
  - Navigation operator (.)
- Arithmetic operators:
  - * (multiplication), / (division), + (addition) and - (subtraction).
- Comparison operators:
  - =, <>, <, <=,>, >=, IS [NOT] NULL, [NOT] BETWEEN,
- Logical operators:
  - AND, OR, NOT.

Criteria query

- Criteria API is a programmatic approach to query instead of string-based approach as in JPQL.
- Good for Dynamic queries.

JPQL

```
Query query =
   entityManager.createQuery("select m
   from Member m where m.memberNumber
   =:number");
```

Criteria API

```
CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder
();

CriteriaQuery<Member> query=

criteriaBuilder.createQuery(Member.class);

Root<Member> memberRoot = query.from(Member.class);

query.select(memberRoot);

query.where(criteriaBuilder.equal(memberRoot.get("memberNumber"),
    number) );
```

EXAMPLES

```
@Query(value = "SELECT e FROM Employee e WHERE e.lastName = :lastname")
public List<Employee> findByLastName(String lastname);
```

Main points

- Spring provides a Transactional capability for ORM applications.

- The mechanism of transcending allows the individual to tap into Transcendental Consciousness and enlivens its qualities in activity.