JPA & Spring Data JPA Annotations — Full Overview with Practical Code Examples

Entity and Table Mapping

`@Entity`

Marks a class as a JPA entity (maps to a database table).

```
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
}
```

Use on every domain model class you want as a DB table.

`@Table`

Customizes the DB table used for the entity.

```
@Entity
@Table(name = "products") // Table will be named "products" in the DB
public class Product { ... }
```

Use when you want to explicitly set the table name or schema.

`@Id` **and** `@GeneratedValue`

Marks the primary key and specifies how it is generated.

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
}
```

Use on your PK field. `GenerationType.IDENTITY` *is common for auto-increment.*

`@Column`

Provides column details (name, length, nullable, etc.).

```
@Entity
public class User {
    @Id @GeneratedValue
    private Long id;
    @Column(name = "user_name", nullable = false, length = 100)
    private String username;
}
```

Use to customize the mapping between Java fields and DB columns.

`@Transient`

Prevents a field from being persisted.

```
@Entity
public class Product {
    @Id @GeneratedValue private Long id;
    @Transient
    private int totalViewCount; // not saved to DB
}
```

Use for calculated or temporary fields you don't want in the DB.

Entity Relationships

`@OneToOne`

Defines one-to-one relationship.

```
@Entity
public class User {
    @Id @GeneratedValue private Long id;
    @OneToOne
    private Passport passport;
}
```

Use when each row in table A maps directly to one row in table B (e.g., one User, one Passport).

`@OneToMany` and `@ManyToOne`

Defines one-to-many and many-to-one relationships.

```
@Entity
public class User {
    @Id @GeneratedValue private Long id;
    @OneToMany(mappedBy = "user")
    private List<Order> orders;
}

@Entity
public class Order {
    @Id @GeneratedValue private Long id;
    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;
}
```

Use for parent-child (one-to-many) and child-parent (many-to-one) relationships.

`@ManyToMany` and `@JoinTable`

Defines many-to-many relationship using a join table.

```
@Entity
public class Student {
    @Id @GeneratedValue private Long id;
    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses;
}
```

Use for relationships where records in both tables connect to multiple records in the other (e.g., students and courses).

`@JoinColumn` and `@JoinTable`

Customizes the foreign key column or join table mapping.

```
@Entity
public class Order {
    @Id @GeneratedValue private Long id;
    @ManyToOne
    @JoinColumn(name = "user_id", nullable = false)
    private User user;
}
```

Use `@JoinColumn` on the owning side of relationships to customize FK columns. Use `@JoinTable` for many-to-many join tables.

Embedded & Value Types

`@Embedded` and `@Embeddable`

Embeds a value object within an entity.

```
@Embeddable
public class Address {
    private String street;
    private String city;
}

@Entity
public class Customer {
    @Id @GeneratedValue private Long id;
    @Embedded
    private Address address;
}
```

Use to include reusable value-type components directly in your entity without creating another table.

Enumerations & Large Objects

`@Enumerated`

Maps enums to a database column.

```
public enum Role { ADMIN, USER }

@Entity
public class Employee {
    @Id @GeneratedValue private Long id;
    @Enumerated(EnumType.STRING)
    private Role role;
}
```

Use to control enum persistence as string or ordinal.

`@Lob`

Marks a field for large object (CLOB/BLOB) storage.

```
@Entity
public class Document {
    @Id @GeneratedValue private Long id;
    @Lob
    private String content;      // CLOB
    @Lob
    private byte[] fileData;     // BLOB
}
```

Use for fields storing large texts or files.

Spring Data JPA Repository Layer

`@Repository`

Marks a DAO or repository class.

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {}
```

Spring Boot auto-detects repositories. Use for custom repository components if not extending.

`@Query` **&** `@Param`

For custom queries in repositories.

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Query("SELECT u FROM User u WHERE u.status = 1")
    List<User> findAllActiveUsers();

    @Query("SELECT u FROM User u WHERE u.username = :username")
    User findByUsername(@Param("username") String username);

    @Query(value = "SELECT * FROM users WHERE status = 1", nativeQuery = true)
    List<User> findActiveUsersNative();
}
```

Use `@Query` for JPQL or native SQL. Use `@Param` to bind parameters.

`@Modifying` **&** `@Transactional`

For custom update or delete queries, and transaction management.

```
@Transactional
@Modifying
@Query("UPDATE User u SET u.status = :status WHERE u.id = :id")
void updateUserStatus(@Param("id") Long id, @Param("status") int status);
```

Use on repository methods when executing DML statements that change data.

`@Procedure`

Maps repository methods to database stored procedures.

```
@Procedure("update_user_status")
void updateUserStatus(Long id, String newStatus);
```

Use to call stored procedures directly from repository interfaces.

Summary Table: Annotation Usage

Annotation	Where / When to Use	Example
`@Entity`	On every model mapped to a table	`public class Product { ... }`
`@Table`	To customize table name	`@Table(name="products")`
`@Id`	On the PK field of an entity	`@Id private Long id;`
`@GeneratedValue`	On PK—auto generating values	`@GeneratedValue(...)`
`@Column`	To customize columns	`@Column(name="name")`
`@Transient`	For non-persistent fields	`@Transient private int temp;`
`@OneToOne`	For one-to-one relations	`@OneToOne private Passport passport;`
`@OneToMany`	For one-to-many relations	`@OneToMany(mappedBy="user")`
`@ManyToOne`	For many-to-one relations	`@ManyToOne` `@JoinColumn(name="user_id")`
`@ManyToMany`	For many-to-many relations	`@ManyToMany @JoinTable(...)`
`@JoinColumn`	FK customization	`@JoinColumn(name="user_id")`
`@Embedded`	Embed value type objects in entity	`@Embedded private Address addr;`
`@Embeddable`	For value type class	`@Embeddable class Address`
`@Enumerated`	Enum mapping	`@Enumerated(EnumType.STRING)`
`@Lob`	For large objects in DB	`@Lob private String data;`

Annotation	Where / When to Use	Example
`@Repository`	On custom repository classes	`@Repository` on interface
`@Query`	For custom queries	`@Query("SELECT ...")`
`@Param`	Bind query params in custom queries	`@Param("username") String username`
`@Modifying`	For update or delete @Query methods	`@Modifying @Query(...)`
`@Transactional`	Transaction boundaries for methods	`@Transactional` on service/repository
`@Procedure`	Call stored procedures	`@Procedure("proc_name") void ...`

These **annotations** are fundamental for Java persistence and data handling in modern Spring Boot/JPA projects. Combine them to efficiently model, query, and persist your data in enterprise and web applications.

[1] [2] [3] [4] [5] [6]

⁂

1. https://www.codingshuttle.com/blogs/top-10-jpa-annotations-in-spring-boot/
2. https://www.baeldung.com/spring-data-jpa-query
3. https://www.geeksforgeeks.org/advance-java/jpa-creating-an-entity/
4. https://www.geeksforgeeks.org/java/spring-data-jpa-query-annotation-with-example/
5. https://www.digitalocean.com/community/tutorials/jpa-hibernate-annotations
6. https://www.linkedin.com/pulse/hibernatejpa-commonly-used-annotations-aqeel-abbas