JPA & Hibernate: Key Annotations

When you start learning and using Hibernate and JPA, the number of annotations might be overwhelming. But as long as you rely on the defaults, you can implement your persistence layer using only a small subset of them.

Let's take a look at the most important annotations and their attributes. For each annotation, I will explain which attributes you really need and which ones you should better avoid.

And if you want to dive deeper into JPA and make sure you have a solid understanding of all the basic concepts, I recommend enrolling in my JPA for Beginners online course.

Define an Entity Class

JPA entities don't need to implement any interface or extend a superclass. They are simple POJOs. But you still need to identify a class as an entity class, and you might want to adapt the default table mapping.

@Entity

The JPA specification requires the @Entity annotation. It identifies a class as an entity class.

```
@Entity
public class Author { ... }
```

Attributes	
name	Unique name of the entity by which it gets referenced in JPQL queries

@Table

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

```
@Entity
@Table(name = "AUTHORS", schema = "STORE")
public class Author {
```

Attributes	
name	Name of the database table.
Schema	Name of the database schema in which the table is located.
catalog	Name of the database catalog that stores the metadata information of the table.
indexes	Index definitions used in CREATE TABLE statement. Not recommended to be used.
uniqueConstraints	Unique constraints used in CREATE TABLE statement. Not recommended to be used.

Basic Column Mappings

By default, all JPA implementations map each entity attribute to a database column with the same name and a compatible type. The following annotations enable you to perform basic customizations of these mappings. You can, for example, change the name of the column, adapt the type mapping, identify primary key attributes, and generate unique values for them.

@Column

Let's start with the *@Column* annotation. It is an optional annotation that enables you to customize the mapping between the entity attribute and the database column.

```
@Entity
public class Book {

    @Column(name = "title", updatable = false, insertable = true)
    private String title;

    ...
}
```

Attributes	
name	Name of the database column.
insertable	Set to false to exclude from SQL INSERT statements.
updatable	Set to false to exclude from SQL UPDATE statements.
table	Only used when you map your entity to 2 database tables.

	Specifies the table that contains the mapped column.
columnDefinition	SQL fragment that's used during table definition. Not recommended to be used.
length	Length of a String-valued database column Not recommended to be used.
scale	Scale of a decimal column. Not recommended to be used.
precision	Precision of a decimal column. Not recommended to be used.
unique	Defines a unique constraint on the mapped column. Not recommended to be used.

@Id

JPA and Hibernate require you to specify at least one primary key attribute for each entity. You can do that by annotating an attribute with the *@Id* annotation.

```
@Entity
public class Author {

    @Id
    private Long id;

    ...
}
```

@GeneratedValue

When we're talking about primary keys, we also need to talk about sequences and auto-incremented database columns. These are the 2 most common database features to generate unique primary key values.

If you annotate your primary key attribute with the *@GeneratedValue* annotation, you can use a database sequence by setting the strategy attribute to *GenerationType.SEQUENCE*. Or, if you want to use an auto-incremented database column to generate your primary key values, you need to set the strategy to *GenerationType.IDENTITY*.

```
@Entity
public class Author {

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

    ...
}
```

Attributes	
strategy	The strategy used to generate the primary key value: • GenerationType.SEQUENCE • GenerationType.IDENTITY
generator	References a custom generator configuration

@Enumerated

The *@Enumerated* annotation enables you to define how anenum attribute gets persisted in the database. By default, all JPA implementations map the ordinal value of the enum to a numeric database column.

As I explained in more detail in my guide on enum mappings, the ordinal makes it hard to add or remove values to the enum. The mapping as a String is more robust and much easier to read. You can activate this mapping by *EnumType.STRING* to the *@Enumerated* annotation.

```
@Entity
public class Author {

    @Enumerated(EnumType.STRING)
    private AuthorStatus status;

    ...
}
```

Attributes	
value	The mapping strategy used for the enum: • EnumType.STRING • EnumType.ORDINAL

@Temporal

If you're still using *java.util.Date* or *java.util.Calendar* as your attribute types, you need to annotate the attribute with *@Temporal*. Using this annotation, you can define if the attribute shall be mapped as an SQL DATE, TIME, or TIMESTAMP.

```
@Entity
public class Author {

    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;

    ...
}
```

Attributes	
value	The SQL type to which the attribute shall be mapped: • TemporalType.DATE • TemporalType.TIME • TemporalType.TIMESTAMP

@Lob

Using JPA's *@Lob*annotation, you can map a BLOB (binary large object) to a *byte[]* and a CLOB (character large object) to a *String*. Your persistence provider then fetches the whole BLOB or CLOB when it initializes the entity attribute.

```
@Entity
public class Book {

    @Lob
    private byte[] cover;

    ...
}
```

In addition to that, Hibernate also supports mappings to *java.sql.Blob* and *java.sql.Clob*. These are not as easy to use a *byte[]* or a *String*, but they can provide better performance. I explained that mapping in great detail in Mapping BLOBs and CLOBs with Hibernate and JPA.

Association Mappings

You can also map associations between your entities. In the table model, these are modeled as foreign key columns. These associations are mapped as attributes of the type of the associated entity or a *Collection* of associated entities, in your domain model.

In both cases, you need to describe the association mapping. You can do that using a *@ManyToMany*,*@ManyToOne*, *@OneToMany*, or *@OneToOne* annotation.

@ManyToMany

Many-to-many associations are very common in relational table models. In your domain model, you can map this association in a uni- or bidirectional way using attributes of type *List*,*Set* or *Map*, and a *@ManyToMany* annotations.

```
@Entity
@Table(name = "BOOKS")
public class Book {

    @ManyToMany
    private Set<Author> authors;

    ...
}
```

Attributes	
fetch	Shall the association get fetched eagerly or lazily.

cascade	The operations that shall be cascaded to the associated entities.

If you want to model the association in a bidirectional way, you need to implement a similar mapping on the referenced entity. But this time, you also need to set the *mappedBy* attribute of the *@ManyToMany* annotation to the name of the attribute that owns the association. To your persistence provider, this identifies the mapping as a bidirectional one.

```
@Entity
public class Author {

    @ManyToMany(mappedBy = "authors")
    private Set<Book> books;

    ...
}
```

You use the same @ManyToMany annotation to define the referencing side of the association, as you use to specify the owning side of it. So, you can use the same *cascade* and *fetch* attributes, as I described before.

@ManyToOne and @OneToMany

Many-to-one and one-to-many associations represent the same association from 2 different perspectives. So, it's no surprise that you can use them together to define a bidirectional association. You can also use each of them on their own to create a unidirectional many-to-one or one-to-many association. But you should avoid unidirectional one-to-many associations. Hibernate handles them very inefficient.

@ManyToOne

Let's take a closer look at the *@ManyToOne* annotation. It defines the owning side of a bidirectional many-to-one/one-to-many

association. You do that on the entity that maps the database table that contains the foreign key column.

```
@Entity
public class Book {

    @ManyToOne(fetch = FetchType.LAZY)
    private Publisher publisher;

    ...
}
```

Attributes	
fetch	Shall the association get fetched eagerly or lazily.
cascade	The operations that shall be cascaded to the associated entities.
optional	Indicates if this association is mandatory

@OneToMany

Similar to the referencing side of a bidirectional many-to-many association, you can reference the name of the attribute that owns the association in the *mappedBy* attribute. That tells your persistence provider that this is the referencing side of a bidirectional association, and it reuses the association mapping defined by the owning side.

```
@Entity
public class Publisher {

    @OneToMany(mappedBy = "publisher", cascade = CascadeType.ALL)
    private Set<Book> books;

    ...
}
```

Attributes	
fetch	Shall the association get fetched eagerly or lazily.
cascade	The operations that shall be cascaded to the associated entities.
orphanRemoval	Shall child entities get removed when removed from association

@OneToOne

One-to-one associations are only rarely used in relational table models. You can map them using a*@OneToOne* annotation.

Similar to the previously discussed association mapping, you can model a uni- or bidirectional one-to-one associations. The attribute that's defined on the entity that maps the database table that contains the foreign key column owns the association.

```
@Entity
public class Manuscript {

    @OneToOne(fetch = FetchType.LAZY)
    private Book book;

    ...
}
```

Attributes	
fetch	Shall the association get fetched eagerly or lazily.
cascade	The operations that shall be cascaded to the associated entities.

optional	Indicates if this association is mandatory

And if you model it as a bidirectional association, you need to set the *mappedBy* attribute of the referencing side of the association to the attribute name that owns the association.

```
@Entity
public class Book {

    @OneToOne(mappedBy = "book")
    private Manuscript manuscript;

    ...
}
```