Theory: Introduction to JPA

828 users solved this topic. Latest completion was about 20 hours ago.

13 minutes 0 / 5 problems solved

Skip this topic Start practicing

§1. What is Persistence?

Persistence is a concept referring to data that stays alive even after the death of the process that had created this data. Persistence can be achieved by storing your data.

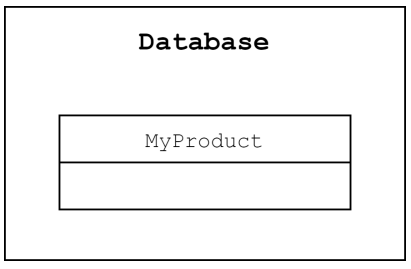
Imagine that you are developing an application that creates a shopping list, and you decided to keep your products inside an array, list, or any other Java object. When you terminate your app, all Java objects, including your shopping list, will be removed from the memory, and you will lose all products from the list! To prevent it, we can save the shopping list, for example, to a database, so after restarting or termination, the app shopping list will not be erased. Such data are called "persistent".

Java Persistence API (JPA) is a specification that describes how to make data persistent in several ways. The most popular way is saving data to relational databases. Using JPA, we can connect **relational databases** and object-oriented models to store, update, and retrieve data from the database through Java objects and vice versa. Such mapping is called **Object-relational mapping** (ORM). So JPA is a specification for ORM, however, it needs implementation. Several frameworks, such as Hibernate or TopLink, implement JPA.

§2. JPA Entity

For representing data from a database in Java objects model, we need **entities**. An entity is a POJO (Plain Old Java Object) class, i.e. an ordinary Java class that should not extend prespecified classes and implement prespecified interfaces. The entity represents a database table, while fields of the entity represent table columns, and each entity instance represents a table row. Entity class **should have no-argument constructor**, but it may have other constructors. It arises from the fact that JPA has to instantiate your object dynamically. It is not possible without a no-argument constructor. The `@Entity` annotation is placed on top of the entity class and declares that a class should be persisted in a database.

Let's see an example. Imagine you have a `"MyProduct"` table in the database:



Here's the corresponding entity:

```
@Entity
public class MyProduct {
}
```

By default, the POJO class with `@Entity` annotation corresponds to the table with the **same name as the class name**. If you want to bind the entity with a table that has a **different name**, you can use `@Table` annotation with the "name = " parameter.

```
@Entity
@Table(name = "MyProduct")
public class Product {
}
```

§3. Persistent Fields

Current topic:

Introduction to JPA

Topic depends on:

- ✗ Introduction to databases
- ✗ Object-Relational Mapping(ORM)
- ✓ Constructor (Stage 6)
- ✓ Annotations
- ✗ Basic project structure

Topic is required for:

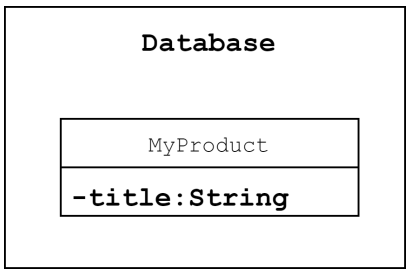
- JPA Relationships
- CRUD Repositories

Table of contents:

↑ Introduction to JPA
§1. What is Persistence?
§2. JPA Entity
§3. Persistent Fields
§4. Primary key
§5. Conclusion
Feedback & Comments

As was mentioned earlier, entity fields represent table columns. It means that each field of the entity maps to the table column. Each entity field **should be private, protected, or package-private**. A field must be **non-static** and **non-final** to be persistent. By default, the name of the entity field corresponds to the column with a similar name.

Let's add `"title"` field to our `"MyProduct"` table:



The corresponding entity looks like the following now:

```
@Entity
public class MyProduct {

    private String title;
}
```

If you want to bind the entity field with a table column that has a **different name**, you can use `@Column` annotation with the "name = " parameter. This annotation is placed on top of the entity field.

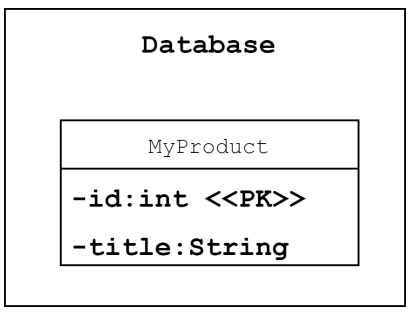
```
@Entity
public class MyProduct {

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

There is another way to bind table columns to the entity. To do that, we can use **getters and setters**. By default, getters' and setters' names should be declared using Java Bean naming conventions. Similarly to entity fields, we can use `@Column` annotation for getters. But let's concentrate on fields for now.

§4. Primary key

Each entity must have a unique identifier or primary key, which is similar to a table's **primary key**. It allows us to identify a specific entity instance. A primary key is also a column, so everything that you learned about binding table columns to entities fields and properties is applicable for the entity primary key, except that we also should use the `@Id` annotation.

Let's add a primary key called `"id"` to our `"MyProduct"` table:



Here's the corresponding entity:

```
@Entity
public class MyProduct {

    @Id
    private int id;

    private String title;
}
```

Note that you can use `@Id` annotation together with `@Column` annotation

```java
@Entity
public class MyProduct {

    @Id
    @Column(name = "id")
    private int productId;

    private String title;
}
```

§5. Conclusion

JPA is a convenient way to deal with relational databases in Java applications. You have learned how to make a Java object persistent using `@Entity` annotation, how to deal with a database table which name is different to Java class name via `@Table` annotation.

Also, now you can map entity fields and properties to the relational data and use `@Column` annotation if the entity field's name differs from the table column name.

Usually, a database table has a primary key that uniquely identifies each row in the table. Using `@Id` annotation, you can specify the primary key property or field of the entity.

Report a typo

76 users liked this theory. **1** didn't like it. **What about you?**

Start practicing

Comments (0) Hints (0) Useful links (0) Show discussion