Defining Unique Constraints in JPA

Last modified: June 3, 2021

by baeldung (https://www.baeldung.com/author/baeldung/)

Persistence (https://www.baeldung.com/category/persistence/)

Hibernate (https://www.baeldung.com/tag/hibernate/)

JPA (https://www.baeldung.com/tag/jpa/)

1. Introduction

In this tutorial, we'll discuss **defining unique constraints using JPA and Hibernate**.

First, we'll discuss unique constraints and how they differ from primary key constraints.

Next, we'll take a look at JPA's important annotations – @*Column(unique=true)* and *@UniqueConstraint.* We'll implement them to define the unique constraints on a single column and multiple columns.

Finally, we'll see how to define unique constraints on referenced table columns.

2. Unique Constraints

Let's start with a quick recap. A unique key is a set of single or multiple columns of a table that uniquely identify a record in a database table.

Both the unique and primary key constraints provide a guarantee for uniqueness for a column or set of columns.

2.1. How It's Different From Primary Key Constraints?

Unique constraints ensure that the data in a column or combination of columns is unique for each row. A table's primary key, for example, functions as an implicit unique constraint. Hence, **the primary key constraint automatically has a unique constraint.**

Furthermore, we can have only one primary key constraint per table. However, there can be multiple unique constraints per table.
Simply put, the unique constraints apply in addition to any constraint entailed by primary key mapping.

The unique constraints we define are used during table creation to generate the proper database constraints and may also be used at runtime to order *insert*, *update*, or *delete* statements.

2.2. What Are Single-Column and Multiple-Column Constraints?

A unique constraint can be either a column constraint or a table constraint. At the table level, we can define unique constraints across multiple columns.

JPA allows us to define unique constraints in our code using *@Column(unique=true)* and *@UniqueConstraint*. These annotations are interpreted by the schema generation process, creating constraints automatically.

Before anything else, let's emphasize that **column-level constraints apply to a single column, and table-level constraints apply to the whole table.**

We'll discuss those in detail in the next sections through examples.

3. Set Up an Entity

An entity (/jpa-entities) in JPA represents a table stored in a database. Every instance of an entity represents a row in the table.

Let's start by creating a domain entity and mapping it to a database table. For this example, we'll create a *Person* entity:

(https://freestar.com/?

```
@Entity
@Table
public class Person implements Serializable {
    @Id
    @GeneratedValue
    private Long id;
    private String name;
    private String password;
    private String email;
    private Long personNumber;
    private Boolean isActive;
    private String securityNumber;
    private String departmentCode;
    @JoinColumn(name = "addressId", referencedColumnName = "id")
    private Address address;
   //getters and setters
 }
```

An *address* field is a referenced field from the *Address* entity:

```
@Entity
@Table
public class Address implements Serializable {
    @Id
    @GeneratedValue
    private Long id;
    private String streetAddress;
    //getters and setters
}
```

Throughout this tutorial, we'll use this *Person* entity to demonstrate our examples.

4. Column Constraints

When we have our model ready, we can implement our first unique constraint.

Let's consider our *Person* entity that holds the person's information. We have a primary key for the *id* column. This entity also holds the *PersonNumber* that does not contain any duplicate value. Also, we can't define the primary key because our table already has it.

In this case, we can use column unique constraints to make sure that no duplicate values are entered in a *PersonNumber* field. JPA allows us to achieve that using the *@Column* annotation with the *unique* attribute.

In this section, we will first take a look at the *@Column* annotation and then learn how to implement it.

4.1. *@Column(unique=true)*

The annotation type *Column* (https://docs.jboss.org/hibernate/jpa/2.1/api/javax/persistence/Column.html) is used to specify the mapped column for a persistent property or field.

Let's take a look at the definition:

```
@Target(value={METHOD,FIELD})
@Retention(value=RUNTIME)
public @interface Column {
    boolean unique;
   //other elements
 }
```

The *unique* attribute specifies whether the column is a unique key. This is a shortcut for the *UniqueConstraint* annotation and is useful when the unique key constraint corresponds to only a single column.

We'll see how to define it in the next section.

4.2. Defining the Column Constraints

Whenever the unique constraint is based only on one field, we can use *@Column(unique=true)* on that column.

Let's define a unique constraint on the *personNumber* field:

```
@Column(unique=true)
private Long personNumber;
```

When we execute the schema creation process, we can validate it from the logs:

```
[main] DEBUG org.hibernate.SQL -
    alter table Person add constraint UK_d44q5lfa9xx370jv2k7tsgsqt unique
(personNumber)
```

Similarly, if we want to restrict a *Person* to register with a unique email, we can add a unique constraint on the *email* field:

```
@Column(unique=true)
private String email;
```

Let's execute the schema creation process and check the constraints:

```
[main] DEBUG org.hibernate.SQL -
    alter table Person add constraint UK_585qcyc8qh7bg1fwgm1pj4fus unique
(email)
```

Although this is useful when we want to put a unique constraint on a single column, sometimes we may want to add unique constraints on a composite key — some combination of columns. To define a composite unique key, we can use table constraints. We'll discuss that in the next section.

5. Table Constraints

A composite unique key is a unique key made up of a combination of columns. **To define a composite unique key, we can add constraints on the table instead of a column. JPA helps us to achieve it using *@UniqueConstraint* annotation.**

5.1. *@UniqueConstraint* Annotation

Annotation type *UniqueConstraint* (https://docs.jboss.org/hibernate/jpa/2.1/api/javax/persistence/UniqueConstraint.html) specifies that a unique constraint is to be included in the generated DDL (Data Definition Language) for a table.

Let's take a look at the definition:

```
@Target(value={})
@Retention(value=RUNTIME)
public @interface UniqueConstraint {
    String name() default "";
    String[] columnNames();
}
```

As we see, **the *name* and *columnNames* of type *String* and *String[]*, respectively, are the annotation elements that may be specified for the *UniqueConstraint* annotation.**

We'll take a better look at each of the parameters in the next section, going through examples.

5.2. Defining Unique Constraints

Let's consider our *Person* entity. A *Person* should not have any duplicate record for the active status. In other words, there won't be any duplicate values for the key comprising *personNumber* and *isActive*. Here, we need to add unique constraints that span across multiple columns.

JPA helps us to achieve that with the *@UniqueConstraint* annotation. We do that in the *@Table* (https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Table.html) annotation under the *uniqueConstraints* attribute. Let's remember to specify the names of the columns:

```
@Table(uniqueConstraints = { @UniqueConstraint(columnNames = {
"personNumber", "isActive" }) })
```

We can validate it once the schema is generated:

(https://freestar.com/?

```
[main] DEBUG org.hibernate.SQL -
    alter table Person add constraint UK5e0bv5arhh7jjhsls27bmqp4a unique
(personNumber, isActive)
```

One point to note here is that if we don't specify a name, it's a provider-generated value. Since JPA 2.0, we can provide a name for our unique constraint:

```
@Table(uniqueConstraints = { @UniqueConstraint(name =
"UniqueNumberAndStatus", columnNames = { "personNumber", "isActive" }) })
```

And we can validate the same:

```
[main] DEBUG org.hibernate.SQL -
    alter table Person add constraint UniqueNumberAndStatus unique
(personNumber, isActive)
```

Here, we have added unique constraints on a set of columns. We can also add multiple unique constraints — unique constraints on multiple sets of columns. We'll do just that in the next section.

5.3. Multiple Unique Constraints on a Single Entity

A table can have multiple unique constraints. In the last section, we defined unique constraints on a composite key: *personNumber* and *isActive* status. In this section, we'll add constraints on the combination of *securityNumber* and *departmentCode*.

Let's collect our unique indexes and specify them at once. We do that by repeating *@UniqueConstraint* annotation in braces and separated by a comma:

```
@Table(uniqueConstraints = {
   @UniqueConstraint(name = "UniqueNumberAndStatus", columnNames =
{"personNumber", "isActive"}),
   @UniqueConstraint(name = "UniqueSecurityAndDepartment", columnNames =
{"securityNumber", "departmentCode"})})
```

Now, let's see the logs and check the constraints:

```
[main] DEBUG org.hibernate.SQL -
    alter table Person add constraint UniqueNumberAndStatus unique
(personNumber, isActive)
[main] DEBUG org.hibernate.SQL -
   alter table Person add constraint UniqueSecurityAndDepartment unique
(securityNumber, departmentCode)
```

Up until now, we defined unique constraints on the fields in the same entity. However, in some cases, we may have referenced fields from other entities and need to ensure the uniqueness of those fields. We'll discuss that in the next section.

6. Unique Constraints on a Referenced Table Column

When we create two or more tables that are related to each other, they are often related by a column in one table referencing the primary key of the other table. That column is called the "foreign key". For example, *Person* and *Address* entities are connected through the *addressId* field. Hence, *addressId* acts as a referenced table column.

We can define unique constraints on the referenced columns. We'll first implement it on a single column and then on multiple columns.

6.1. Single-Column Constraints

In our *Person* entity, we have an *address* field that refers to the *Address* entity. A *person* should have a unique address.

So, let's define a unique constraint on the *address* field of the *Person*:

```
@Column(unique = true)
private Address address;
```

Now, let's quickly check this constraint:

```
[main] DEBUG org.hibernate.SQL -
   alter table Person add constraint UK_7xo3hsusabfaw1373oox9uqoe unique
(address)
```

We can also define multiple column constraints on the referenced table column, as we'll see in the next section.

6.2. Multiple-Column Constraints

We can specify unique constraints on a combination of columns. As stated earlier, we can use table constraints to do so.

Let's define unique constraints on the *personNumber* and *address* and add it to the *uniqueConstraints* array:

```
@Entity
@Table(uniqueConstraints =
  //other constraints
  @UniqueConstraint(name = "UniqueNumberAndAddress", columnNames = {
"personNumber", "address" })})
```

Finally, let's see the unique constraints:

```
[main] DEBUG org.hibernate.SQL -
    alter table Person add constraint UniqueNumberAndAddress unique
(personNumber, address)
```

7. Conclusion

The unique constraints prevent two records from having identical values in a column or set of columns.

In this tutorial, we saw how we could define unique constraints in JPA. First, we did a little recap of the unique constraints. Further, we discussed *@Column(unique=true)* and *@UniqueConstraint* annotations to define unique constraints on a single column and multiple columns, respectively.

As always, the examples from the article are available over on GitHub (https://github.com/eugenp/tutorials/tree/master/persistence-modules/java-jpa-3).

Comments are closed on this article!