Spring Data JPA/Hibernate – Primary Key Generation Strategies

Description

Introduction

In this particular post we will see the different ID generation strategies present in JPA. After reading this post you will be familiar and comfortable understanding the various strategies. Also this is one of the most important interview questions asked across different experience levels. First we'll give a short introduction on various ID generation strategies and then we'll have a look at each one using simple and easy to understand examples. Using the appropriate Id generation strategy is also very important based on the use case.

Simple Identifiers

This is one of the simplest way to define a class attribute as primary key by using @Id annotation. The developer is responsible to set the value manually each time while persisting the record to the database.


```
@Entity
@Table(name="unicorn")
@Data
public class UnicornEntity {

    @Id
    private Integer id;

    // ...
}
```

Generated Identifiers

As we saw above, the @Id annotation specifies the primary key of an entity and the @GeneratedValue specify the primary key generation strategy.

First we will see different ID generation strategies that are present and then we will go over each one using an example. There are four ID generation types in JPA: AUTO, IDENTITY, SEQUENCE and TABLE. AUTO is the default type when you don't specify any generation strategy. @Id and @GeneratedValue are the basic annotations used above the id attribute. There are additional annotations, which we use based on the generation type.

AUTO generation type

AUTO is the default generation type. It is the responsibility of persistence provider to figure out the value based on the datatype provided for the primary key attribute. AUTO picks any of the other three (IDENTITY, SEQUENCE and TABLE) strategy based on the underlying database capabilities.

```
@Entity
@Table(name="unicorn")
@Data
public class UnicornEntity {

    @Id
    @GeneratedValue
    private Integer id;

    // ...
}
```

If we want to use UUID as the primary key then we need to change the type of the primary key.


```
@Entity
@Table(name="unicorn")
@Data
public class UnicornEntity {

    @Id
    @GeneratedValue
    private UUID id;

    // ...
}
```

*IDENTITY*Â generation type

The IDENTITY generation type depends on the IdentityGenerator feature which expects the database to provide value for the identity column. This generation type is most suited for databases which has an auto-incremented column (E.g. AUTO_INCREMENT feature in MySQL and IDENTITY in SQL server). For JPA and Hibernate, we should mostly try SEQUENCE generation typeÂ if the relational database supports it. The reason is because Hibernate fails to use automatic JDBC batching when persisting entities using the IDENTITY generator. In short IDENTITY generation type does not support batch updates.

```
@Entity
@Table(name="unicorn")
```

```
@Data
public class UnicornEntity {

    @Id
    @GeneratedValue(
            strategy = GenerationType.IDENTITY,
            generator = "id_generator"
            )
    private Integer id;

    // ...
}
```

SEQUENCE generation type

A database sequence object generates the identifier values. This can be considered as the best generation strategy.

If you look at the code below, inside the **@GeneratedValue** annotation you will see a property named generator, which has a value of **“id_generator”**. This refers to the **@SequenceGenerator** annotation which has **name = “id_generator”** property and few other properties.


```
@Entity
@Table(name="unicorn")
@Data
public class UnicornEntity {

    @Id
    @GeneratedValue(
            strategy = GenerationType.SEQUENCE,
            generator = "id_generator"
    )
    @SequenceGenerator(
            name = "id_generator",
            sequenceName = "id_generator_name",
            allocationSize = 1
    )
    private Integer id;

    // ...
}
```

Once you run your application, you will see a table named **id_generator_name** will get created and initial value will be 1.

```
    Hibernate: create table id_generator_name (next_val bigint) engine=MyISAM
    Hibernate: insert into id_generator_name values ( 1 )
```

*TABLE*Â generation type

The TABLE generator strategy does not have any direct implementation in relational databases. Whereas databases have implementation for SEQUENCE (auto_increment column) and IDENTITY (identifier generation) generation types. Here we use a table as id generation table.

The @TableGenerator annotation is used to create a table and set the initial value for the id value.

Then it uses the table to do the value generation in the @GeneratedValue annotation.

```
@Entity
@Table(name="unicorn")
@Data
public class UnicornEntity {

    @TableGenerator(
            name = "id_generator",
            table = "id_gen",
            pkColumnName = "gen_name",
            valueColumnName = "gen_val",
            pkColumnValue = "iden_Gen",
            initialValue = 10000,
            allocationSize = 100)
    @Id
    @GeneratedValue(
            strategy = GenerationType.TABLE,
            generator = "id_generator"
    )
    private Integer id;

    // ...
}
```


Custom Generator

Customer Generators are used when we don’t want to use any of the above mentioned strategies. In that way we can use our own custom generator by creating a separate class which implements the **IdentifierGenerator** interface. This custom class will be responsible to generate Identifier values for us.

Twitter Snowflake for ID generation strategy is used by Twitter for their unique Tweet IDs. For customized ID generation methods like these we can use custom generator.

```
@Entity
@Table(name="unicorn")
@Data
```

```
public class UnicornEntity {

    @Id
    @GenericGenerator(name="id_generator", strategy = "com.springmicroservices
    @GeneratedValue
    private Integer id;

    // ...
}
```

```
import org.hibernate.HibernateException;
import org.hibernate.engine.spi.SharedSessionContractImplementor;
import org.hibernate.id.IdentifierGenerator;

public class CustomIdGenerator implements IdentifierGenerator {
    @Override
    public Object generate(SharedSessionContractImplementor sharedSessionContr
        return System.currentTimeMillis();
    }
}
```


Here we have a custom class named CustomIdGenerator which is responsible to generate ID's. For the sake of this example we are using System.currentTimeMillis() method to generate the value.

Composite Identifiers

Composite Identifiers in Hibernate is used to represent the composite primary key in the database. There are some use cases where we cannot identify rows in a table uniquely using a single column, in such cases we would need multiple columns to define a primary key. Hence if multiple columns are used to define a primary key then its call as Composite key. Hibernate provides annotations to define Composite Identifiers.

@Embeddable and @EmdeddedId

Lets take an example of Student. We have created a StudentPK class which will act as the composite primary key, and we will use this key in the main entity class i.e. StudentEntity class. This class will be annotated with @Embeddable annotation.

```
@Embeddable
@Data
@EqualsAndHashCode
public class StudentPK implements Serializable {

    private Integer rollNumber;
    private String mobile;
}
```

Composite Identifier defined above will be used in StudentEntity class by annotating with @EmbeddedId annotation.

```
@Entity
public class StudentEntity {

    @EmbeddedId
    private StudentPK studentId;

    // ...
}
```

@IdClass

The @IdClass is similar to @EmbeddedId annotation. The main difference is we are using the attributes in the main entity class using @Id for each attribute.


```
@Embeddable
@Data
@EqualsAndHashCode
public class StudentPK implements Serializable {

    private Integer rollNumber;
    private String mobile;
}
```

The above class will remain same as we used in the previous section, the only difference is in the main entity class.

```
@Entity
@IdClass(StudentPK.class)
public class StudentEntity {

    @Id
    private Integer rollNumber;
    @Id
    private String mobile;

    // ...
}
```

Important points to remember while implementing composite keys

- It should be defined using @*EmbeddedId*Â or @IdClassÂ annotations.
- It should be public, serializable and have a public no-arg constructor.
- It should implement *equals()* andÂ *hashCode()*Â methods.

Conclusion

This article showcased various ways of defining identifiers in Hibernate.

Category

1. Hands on
2. Spring Boot

Date Created
May 1, 2023
Author
kk-ravi144gmail-com