Hibernate Inheritance Mapping

Last modified: April 15, 2018

by baeldung (/author/baeldung/)

Persistence (/category/persistence/)

Hibernate (/tag/hibernate/)

1. Overview

Relational databases don't have a straightforward way to map class hierarchies onto database tables.

To address this, the JPA specification provides several strategies:

- *MappedSuperclass* – the parent classes, can't be entities
- Single Table – the entities from different classes with a common ancestor are placed in a single table
- Joined Table – each class has its table and querying a subclass entity requires joining the tables
- Table-Per-Class – all the properties of a class, are in its table, so no join is required

Each strategy results in a different database structure.

Entity inheritance means that we can use polymorphic queries for retrieving all the sub-class entities when querying for a super-class.

Since Hibernate is a JPA implementation, it contains all of the above as well as a few Hibernate-specific features related to inheritance.

In the next sections, we'll go over available strategies in more detail.

2. *MappedSuperclass*

Using the *MappedSuperclass* strategy, inheritance is only evident in the class, but not the entity model.

Let's start by creating a *Person* class which will represent a parent class:

```java
@MappedSuperclass
public class Person {

    @Id
    private long personId;
    private String name;

    // constructor, getters, setters
}
```

Notice that this class no longer has an *@Entity* annotation, as it won't be persisted in the database by itself.

Next, let's add an *Employee* sub-class:

```java
@Entity
public class MyEmployee extends Person {
    private String company;
    // constructor, getters, setters
}
```

In the database, this will correspond to one *"MyEmployee"* table with three columns for the declared and inherited fields of the sub-class.

If we're using this strategy, ancestors cannot contain associations with other entities.

3. Single Table

The Single Table strategy creates one table for each class hierarchy. This is also the default strategy chosen by JPA if we don't specify one explicitly.

We can define the strategy we want to use by adding the *@Inheritance* annotation to the super-class:

```java
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class MyProduct {
    @Id
    private long productId;
    private String name;

    // constructor, getters, setters
}
```

The identifier of the entities is also defined in the super-class.

Then, we can add the sub-class entities:

```java
@Entity
public class Book extends MyProduct {
    private String author;
}
```

```java
@Entity
public class Pen extends MyProduct {
    private String color;
}
```

3.1. Discriminator Values

Since the records for all entities will be in the same table, **Hibernate needs a way to differentiate between them.**

By default, this is done through a discriminator column called *DTYPE* which has the name of the entity as a value.

To customize the discriminator column, we can use the *@DiscriminatorColumn* annotation:

```java
@Entity(name="products")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="product_type",
  discriminatorType = DiscriminatorType.INTEGER)
public class MyProduct {
    // ...
}
```

Here we've chosen to differentiate *MyProduct* sub-class entities by an *integer* column called *product_type.*

Next, we need to tell Hibernate what value each sub-class record will have for the *product_type* column:

```java
@Entity
@DiscriminatorValue("1")
public class Book extends MyProduct {
    // ...
}
```

```java
@Entity
@DiscriminatorValue("2")
public class Pen extends MyProduct {
    // ...
}
```

Hibernate adds two other pre-defined values that the annotation can take: "*null*" and "*not null*":

- *@DiscriminatorValue("null")* – means that any row without a discriminator value will be mapped to the entity class with this annotation; this can be applied to the root class of the hierarchy
- *@DiscriminatorValue("not null")* – any row with a discriminator value not matching any of the ones associated with entity definitions will be mapped to the class with this annotation

Instead of a column, we can also use the Hibernate-specific *@DiscriminatorFormula* annotation to determine the differentiating values:

```java
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorFormula("case when author is not null then 1 else 2 end")
public class MyProduct { ... }
```

This strategy has the advantage of polymorphic query performance since only one table needs to be accessed when querying parent entities. On the other hand, this also means that **we can no longer use *NOT NULL* constraints on sub-class** entity properties.

4. Joined Table

Using this strategy, each class in the hierarchy is mapped to its table. The only column which repeatedly appears in all the tables is the identifier, which will be used for joining them when needed.

Let’s create a super-class that uses this strategy:

```java
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Animal {
    @Id
    private long animalId;
    private String species;

    // constructor, getters, setters
}
```

Then, we can simply define a sub-class:

```java
@Entity
public class Pet extends Animal {
    private String name;

    // constructor, getters, setters
}
```

Both tables will have an *animalId* identifier column. The primary key of the *Pet* entity also has a foreign key constraint to the primary key of its parent entity. To customize this column, we can add the *@PrimaryKeyJoinColumn* annotation:

```java
@Entity
@PrimaryKeyJoinColumn(name = "petId")
public class Pet extends Animal {
    // ...
}
```

The disadvantage of this inheritance mapping method is that retrieving entities requires joins between tables, which can result in lower performance for large numbers of records.

The number of joins is higher when querying the parent class as it will join with every single related child – so performance is more likely to be affected the higher up the hierarchy we want to retrieve records.

5. Table Per Class

The Table Per Class strategy maps each entity to its table which contains all the properties of the entity, including the ones inherited.

The resulting schema is similar to the one using *@MappedSuperclass,* but unlike it, table per class will indeed define entities for parent classes, allowing associations and polymorphic queries as a result.

To use this strategy, we only need to add the *@Inheritance* annotation to the base class:

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Vehicle {
    @Id
    private long vehicleId;

    private String manufacturer;

    // standard constructor, getters, setters
}
```

Then, we can create the sub-classes in the standard way.

This is not very different from merely mapping each entity without inheritance. The distinction is apparent when querying the base class, which will return all the sub-class records as well by using a *UNION* statement in the background.

The use of *UNION* can also lead to inferior performance when choosing this strategy. Another issue is that we can no longer use identity key generation.

6. Polymorphic Queries

As mentioned, querying a base class will retrieve all the sub-class entities as well.

Let's see this behavior in action with a JUnit test:

```
@Test
public void givenSubclasses_whenQuerySuperclass_thenOk() {
    Book book = new Book(1, "1984", "George Orwell");
    session.save(book);
    Pen pen = new Pen(2, "my pen", "blue");
    session.save(pen);

    assertThat(session.createQuery("from MyProduct")
      .getResultList()).hasSize(2);
}
```

In this example, we've created two *Book* and *Pen* objects, then queried their super-class *MyProduct* to verify that we'll retrieve two objects.

Hibernate can also query interfaces or base classes which are not entities but are extended or implemented by entity classes. Let's see a JUnit test using our *@MappedSuperclass* example:

```java
@Test
public void givenSubclasses_whenQueryMappedSuperclass_thenOk() {
    MyEmployee emp = new MyEmployee(1, "john", "baeldung");
    session.save(emp);

    assertThat(session.createQuery(
      "from com.baeldung.hibernate.pojo.inheritance.Person")
      .getResultList())
      .hasSize(1);
}
```

Note that this also works for any super-class or interface, whether it's a *@MappedSuperclass* or not. The difference from a usual HQL query is that we have to use the fully qualified name since they are not Hibernate-managed entities.

If we don't want a sub-class to be returned by this type of query, then we only need to add the Hibernate *@Polymorphism* annotation to its definition, with type *EXPLICIT*:

```java
@Entity
@Polymorphism(type = PolymorphismType.EXPLICIT)
public class Bag implements Item { ...}
```

In this case, when querying for *Items*, the *Bag* records won't be returned.

7. Conclusion

In this article, we've shown the different strategies for mapping inheritance in Hibernate.

The full source code of the examples can be found over on GitHub (https://github.com/eugenp/tutorials/tree/master/hibernate5).

CATEGORIES

SPRING (/CATEGORY/SPRING/)
REST (/CATEGORY/REST/)
JAVA (/CATEGORY/JAVA/)
SECURITY (/CATEGORY/SECURITY-2/)
PERSISTENCE (/CATEGORY/PERSISTENCE/)
JACKSON (/CATEGORY/JACKSON/)
HTTPCLIENT (/CATEGORY/HTTP/)
KOTLIN (/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES/)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES/)

SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT/)
THE COURSES (HTTP://COURSES.BAELDUNG.COM)
CONSULTING WORK (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
CONTACT (/CONTACT)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
EDITORS (/EDITORS)
MEDIA KIT (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)