Localized Data - How to Map It With Hibernate

Let's first take a look at the table model before we discuss how you can map it with JPA and Hibernate. To make that easier to understand, I will use the following example:

We want to create a marketplace in which suppliers can offer their products. The marketplace supports the languages German and English. The supplier can provide the name and description of a product in both languages.

As so often, you can model this in various ways. Shantanu Kher created a great overview of different options and discussed their advantages and disadvantages on the vertabelo blog.

Even though the popularity of these approaches varies, I have seen all of them in real life. In my experience, the most commonly used ones are:

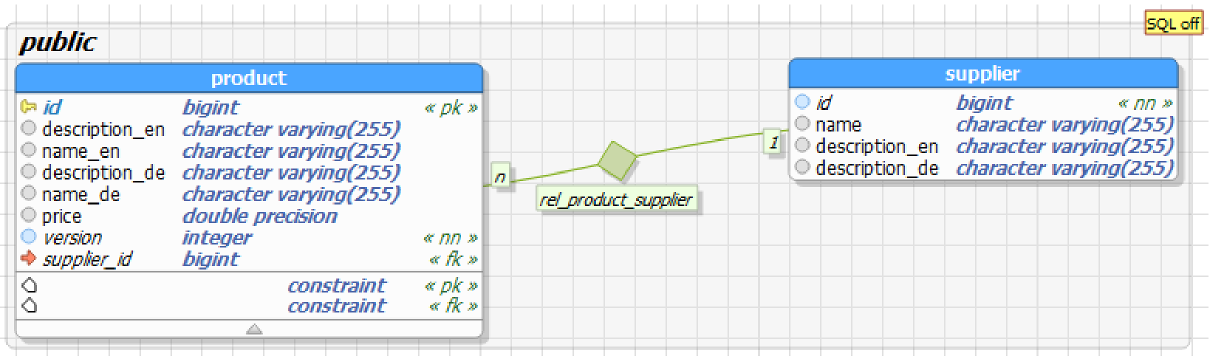
1. Using separate columns for each language in the same database table, e.g., modeling the columns description_en and description_de to store different translations of a product description.
2. Storing translated fields in a separate table. That would move the description_en and description_de columns to a different table. Let's call it LocalizedProduct.

Let's take a closer look at both options.

Separate Language Columns in Each Table

The general idea of this approach is simple. For each localized attribute and language you need to support, you add an extra column to your table. Depending on the number of supported languages and localized attributes, this can result in a vast amount of additional columns. If you want to translate 4 attributes into 5 different languages, you would need to model 4*5=20 database columns.

In the previously described example, you need 4 database columns to localize the product name and description. You use the columns description_en and description_de to persist the different translations of the product description. The columns name_en and name_de to store the localized product name.


public
SQL off
product
id bigint « pk »
description_en character varying(255)
name_en character varying(255)
description_de character varying(255)
name_de character varying(255)
price double precision
version integer « nn »
supplier_id bigint « fk »
constraint « pk »
constraint « fk »
n
rel_product_supplier
1
supplier
id bigint « nn »
name character varying(255)
description_en character varying(255)
description_de character varying(255)

Creating Your Entity Mappings

As you have seen in the previous diagram, using separate columns for each translation results in a straightforward table model. The same is true for the entity mapping.

The id attribute is of type Long and maps the primary key. The @GeneratedValue annotation tells Hibernate to use a database sequence to generate unique primary key values. In this example, I use Hibernate's default sequence. But as I showed in a previous article, you can easily provide your own sequence.

The version attribute is used for optimistic locking and provides a highly scalable way to avoid concurrent updates. I explain it in more details in my Hibernate Performance Tuning Online Training.

The supplier attribute defines the owning side of a many-to-one association to the Supplier entity. As for all to-one associations, you should make sure to set the FetchType to LAZY to avoid unnecessary queries and performance problems.

The nameDe, nameEn, descriptionDe, and descriptionEn attributes just map each of the localized columns. That might result in a lot of attributes, but it is also a simple and efficient way to handle localized data.

```
@Entity
public class Product {

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

    @Version
    private int version;

    @ManyToOne(fetch = FetchType.LAZY)
    private Supplier supplier;

    private Double price;

    @Column(name = "name_de")
    private String nameDe;

    @Column(name = "name_en")
    private String nameEn;

    @Column(name = "description_de")
    private String descriptionDe;

    @Column(name = "description_en")
    private String descriptionEn;

    ...
}
```

Pros & Cons of Entities with Separate Language Columns

As you have seen, adding a dedicated column for each translation to your table:

- Is very easy to implement in the table model,
- Is very easy to map to an entity and
- Enables you to fetch all translations with a simple query that doesn't require any JOIN clauses.

But on the downside:

- this mapping might require a lot of database columns if you need to translate multiple attributes into various languages,
- fetching an entity loads translations that you might not use in your use case and
- you need to update the database schema if you need to support a new language.

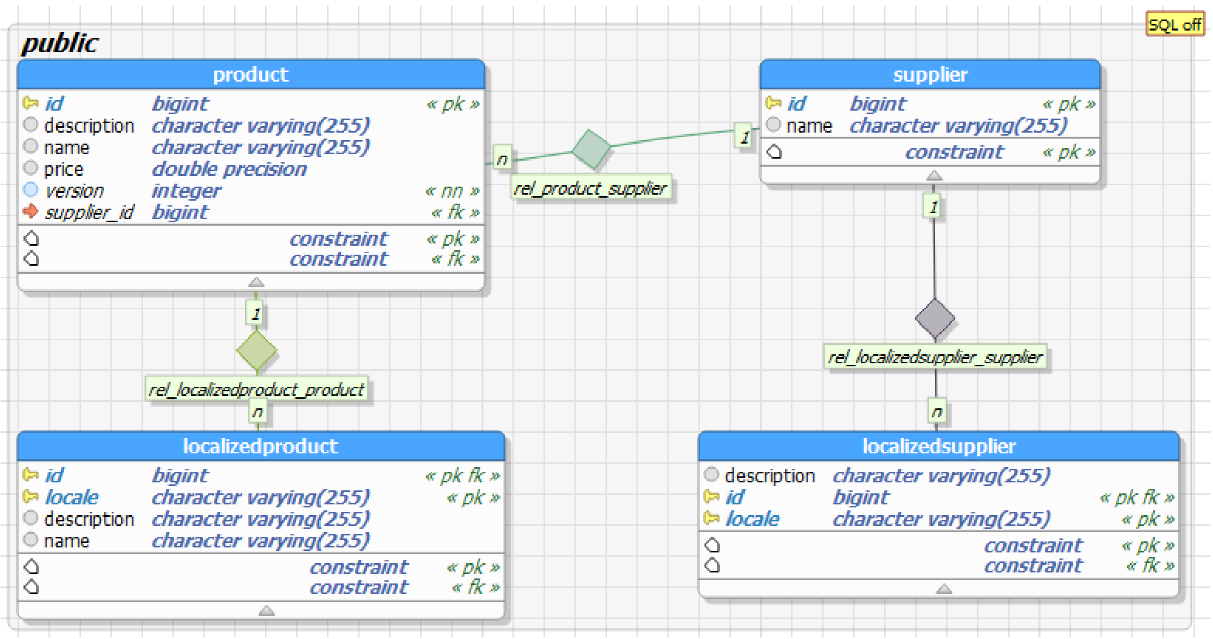
In my experience, the inflexibility of this approach is the biggest downside. If your application is successful, your users and sales team will request additional translations. The required schema update makes supporting a new language much harder than it should be. You not only need to implement and test that change, but you also need to update your database without interrupting your live system.

The next approach avoids these problems, and I, therefore, recommend it for most applications.

Different Tables and Entities for Translated and Non-Translated Fields

Instead of storing all the information in the same database table, you can also separate the translated and non-translated fields into 2 tables. That enables you to model a one-to-many association between the non-translated fields and the different localizations.

Here you can see a table model that applies this approach to the previously discussed example.


SQL off
public
product
id bigint « pk »
description character varying(255)
name character varying(255)
price double precision
version integer « nn »
supplier_id bigint « fk »
constraint « pk »
constraint « fk »
supplier
id bigint « pk »
name character varying(255)
constraint « pk »
rel_product_supplier
n
1
rel_localizedproduct_product
1
n
rel_localizedsupplier_supplier
1
n
localizedproduct
id bigint « pk fk »
locale character varying(255) « pk »
description character varying(255)
name character varying(255)
constraint « pk »
constraint « fk »
localizedsupplier
description character varying(255)
id bigint « pk fk »
locale character varying(255) « pk »
constraint « pk »
constraint « fk »

The LocalizedProduct table stores the different translations of the product name and description. As you can see in the diagram, that table contains a record for each localization of a product. So, if you want to store an English and a German name and description of your product, the LocalizedProduct table contains 2 records for that product. And if you're going to support an additional language, you only need to add another record to the LocalizedProduct table instead of changing your table model.

Creating Your Entity Mappings

The entity model is almost identical to the table model. You map the non-translated columns of the Product table to the Product entity and the translated columns of the LocalizedProduct table to the LocalizedProduct entity. And between these 2 entity classes, you can model a managed many-to-one association.

Entity with Translated Fields - The LocalizedProduct entity

The following mapping of the LocalizedProduct entity consists of a few mandatory and an optional part. Let's first talk about the mandatory mapping of the primary key and the association to the Product entity.

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL)
public class LocalizedProduct {

    @EmbeddedId
    private LocalizedId localizedId;

    @ManyToOne
    @MapsId("id")
    @JoinColumn(name = "id")
    private Product product;

    private String name;

    private String description;

    ...
}
```

The LocalizedProduct entity represents the to-many side of the association. The Product product attribute, therefore, owns the relationship definition. The @JoinColumn annotation tells Hibernate to use the id column of the LocalizedProduct table as the foreign key column. And the @MapsId annotation defines that the primary key value of the associated Product entity is part of the composite primary key of the LocalizedProduct entity. It gets mapped to the id attribute of the primary key class.

As I explain in great details in the Advanced Hibernate Online Training, you can map a composite primary key in various ways with JPA and Hibernate. In this example, I use an embedded id and an embeddable called LocalizedId.

As you can see in the following code snippet, the LocalizedId class is a basic Java class which implements the Serializable interface and is annotated with @Embeddable. And because you want to use it as an embedded id, you also need to make sure to implement the equals and hashCode methods.

```
@Embeddable
public class LocalizedId implements Serializable {

    private static final long serialVersionUID = 1089196571270403924L;

    private Long id;

    private String locale;

    public LocalizedId() {
    }

    public LocalizedId(String locale) {
        this.locale = locale;
    }

    ...
}
```

If you want to take it one step further, you might also want to cache the LocalizedProduct entity. You can do that by activating the cache in your persistence.xml configuration and by annotating the LocalizedProduct entity with JPA's @Cacheable or Hibernate's @Cache annotation. As I explain in my Hibernate Performance Tuning Online Training, caching is a two-edged sword. It can provide substantial performance benefits but also introduce an overhead which can slow down your application. You need to make sure that you only change data that gets often read but only rarely changed. In most applications, that's the case for the localized Strings. That makes them excellent candidates for caching.

Entity with Non-Translated Fields - The Product entity

After we mapped the LocalizedProduct table, which represents the different translations of the localized fields, it's time to work on the mapping of the Product table.

The only difference to the previous example is the mapping of the localized attributes. Instead of mapping an attribute for each translation, I'm using the localizations attribute. It maps the referencing side of the many-to-one association to the LocalizedProduct entity to a java.util.Map. This is one of the more advanced association mappings defined by the JPA specification, and I explained in great details in How to map an association as a java.util.Map.

In this example, I use the locale attribute of the LocalizedProduct entity as the key and the LocalizedProduct entity as the value of the Map. The locale is mapped by the LocalizedId embeddable, and I need to specify the path localizedId.locale in the @MapKey annotation.

The mapping to a java.util.Map makes accessing a specific translation in your business code more comfortable. And it doesn't affect how Hibernate fetches the association from the database. In your JPQL or Criteria Queries, you can use this association in the same way as any other managed relationship.

```
@Entity
public class Product {

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

    @Version
    private int version;

    @ManyToOne(fetch = FetchType.LAZY)
    private Supplier supplier;

    private Double price;

    @OneToMany(mappedBy = "product", cascade = {CascadeType.DETACH,
CascadeType.MERGE, CascadeType.PERSIST, CascadeType.REFRESH},
orphanRemoval = true)
    @MapKey(name = "localizedId.locale")
    @Cache(usage = CacheConcurrencyStrategy.TRANSACTIONAL)
    private Map<String, LocalizedProduct> localizations = new HashMap<>();

    ...

    public String getName(String locale) {
        return localizations.get(locale).getName();
    }

    public String getDescription(String locale) {
        return localizations.get(locale).getDescription();
    }
}
```

If you want to make your entity model more comfortable to use, you could activate orphanRemoval for the association. That is a general best practice for one-to-many associations that model a parent-child relationship in which the child can't exist without its parent. It tells your JPA implementation, e.g., Hibernate, to delete the child entity as soon as its association to the parent entity gets removed. I use it in this example to remove a LocalizedProduct entity as soon as it's no longer associated with a Product entity.

Another thing you could do to improve the usability of your entities is to provide getter methods that return the product name and description for a given locale. If you implement additional getter methods to return a localized name and description, you need to keep in mind that they are accessing a lazily fetched one-to-many association. That triggers an additional SQL statement if the association isn't already fetched from the database. You can avoid that by using a JOIN FETCH clause or an entity graph to initialize the association while loading your Product entity.

And if you activated the 2nd level cache on the LocalizedProduct entity, you should also annotate the localizations attribute with Hibernate's @Cache annotation. That tells Hibernate to cache the association between these 2 entities. If you miss this annotation, Hibernate will execute a query to retrieve the associated LocalizedProduct entities even though they might be already in the cache. That is another example of how complex caching with Hibernate can be. It's also one of the reasons why the Hibernate Performance Tuning Online Training includes a very detailed lecture about it.

Pros & Cons of Different Entities for Translated and Non-Translated Fields

Storing your translations in a separate table is a little more complicated, but it provides several benefits:

- Each new translation is stored as a new record in the LocalizedProduct table. That enables you to store new translations without changing your table model.

- Hibernate's 2nd level cache provides an easy way to cache the different localizations. In my experience, other attributes of an entity, e.g., the price, change more often than the translations of a name or description. It can, therefore, be a good idea to separate the localizations from the rest of the data to be able to cache them efficiently.

But the mapping also has a few disadvantages:

- If you want to access the localized attributes, Hibernate needs to execute an additional query to fetch the associated LocalizedProduct entities. You can avoid that by initializing the association when loading the Product entity.
- Fetching associated LocalizedProduct entities might load translations that you don't need for your use case.

Conclusion

Using additional columns to store the translations of a field, might seem like the most natural and obvious choice. But as I showed you in this article, it's very inflexible. Supporting an additional language requires you to change your table and your domain model.

You should, therefore, avoid this approach and store the translated and non-translated information in 2 separate database tables. You can then map each table to an entity and model a one-to-many association between them.

This approach allows you to add new translations without changing your domain and table model. But the mapping is also a little more complicated, and Hibernate needs to execute an additional query to retrieve the different localizations. You can avoid these queries by activating the 2nd level cache.