With JPA and Hibernate, you can map enums in different ways. You can:

- use the standard mappings to a number or a String,
- create a customized mapping to any basic type supported by Hibernate,
- define a custom mapping to a database-specific type, like PostgreSQLs enum type.

In this article, I will show you how to use all 3 of these options to map the following *Rating* enum. It's used in the *Review* entity to represent the rating of a book.

```
public enum Rating {
    ONE, TWO, THREE, FOUR, FIVE
}
```

JPA & Hibernate Standard Enum Mappings

Enums are a first-class citizen of the Java language and used in most domain models. So, it's no surprise that JPA and Hibernate provide a standard mapping for them.

You can choose between 2 mappings:

1. By default, Hibernate maps an enum to a number. It uses the ordinal value, which is the zero-based position of a value within the definition of the enum. So, the enum value that's defined first gets mapped to 0, the second one to 1 and so on.
   This mapping is very efficient, but there is a high risk that adding or removing a value from your enum will change the ordinal of the remaining values.
2. You can map the enum value to a *String*.
   This mapping is not as efficient, but you can add or remove enum values without any side effects. You just can't rename a value without migrating your database.

You don't need to annotate your entity attribute if you want to store the ordinal value of your enum in the database. Here you can see an example of such a mapping.

```
@Entity
public class Review {

    @Id
    @GeneratedValue
    private Long id;

    private String message;

    private Rating rating;

    ...
}
```

If you instead want to persist the String representation of your enum value, you need to annotate your entity attribute with *@Enumerated(EnumType.STRING)*.

```
@Entity
public class Review {

    @Id
    @GeneratedValue
    private Long id;

    private String message;
```

```
    @Enumerated(EnumType.STRING)
    private Rating rating;

    ...
}
```

Customized Mapping to a Supported Basic Type

The 2 previously shown standard mappings are good enough for most domain models. But they can make a refactoring of your persistence layer harder than it should be. They are also not flexible enough to handle the mapping used by most legacy data models.

In all these situations, a custom mapping to a *String* or a number is the better choice. It gives you the flexibility to implement the mapping used in your legacy database. You can handle the changes in your mapping instead of migrating the database if you need to refactor your enum.

Create a Custom Mapping

Creating a custom mapping is straightforward. Youimplement the *AttributeConverter* interface, annotate the class with *@Converter*, and implement the 2 methods that provide the mapping between the entity and the database representation. If you want to use the converter for all attributes of type *Rating* automatically, you can set the *autoApply* attribute of the *@Converter* annotation to *true*.

In the following example, I map the *Rating* enum to an integer.

```
@Converter
public class RatingAttributeConverter
        implements AttributeConverter<Rating, Integer> {

    @Override
    public Integer convertToDatabaseColumn(Rating attribute) {
        if (attribute == null)
            return null;

        switch (attribute) {
        case ONE:
            return 1;

        case TWO:
            return 2;

        case THREE:
            return 3;

        case FOUR:
            return 4;

        case FIVE:
            return 5;

        default:
            throw new IllegalArgumentException(attribute + " not supported.");
        }
    }

    @Override
    public Rating convertToEntityAttribute(Integer dbData) {
        if (dbData == null)
            return null;

        switch (dbData) {
        case 1:
            return Rating.ONE;

        case 2:
            return Rating.TWO;

        case 3:
            return Rating.THREE;

        case 4:
            return Rating.FOUR;

        case 5:
            return Rating.FIVE;
```

```
        default:
            throw new IllegalArgumentException(dbData + " not supported.");
        }
    }

}
```

Use the AttributeConverter

But before you try to use this mapping, you need to put the right annotations on your entity attribute.

You can't use an *AttributeConverter* on an entity attribute that's annotated with *@Enumerated*. So, you need to remove that annotation from your mapping if you want to use the custom mapping.

If you didn't annotate your converter with *@Converter(autoApply=true)*, you also need to annotate the entity attribute with *@Convert(converter = RatingAttributeConverter.class)*. This tells Hibernate to use the referenced converter when it reads or writes this entity attribute.

```
@Entity
public class Review {

    @Id
    @GeneratedValue
    private Long id;

    private String message;

    @Convert(converter = RatingAttributeConverter.class)
    private Rating rating;

    ...
}
```

Hibernate applies the converter transparently whenever you use the *Review* entity and its *rating* attribute in your business code,

A JPQL or a CriteriaQuery. So, you can use the *rating* attribute in the same way as any other entity attribute.

Customized Mapping to a Database-Specific Enum Type

Some databases, like PostgreSQL, offer custom data types to store enumerations. These data types are similar to the enum type that we know in Java. They define a set of valid values that can be stored in the database column.

```
CREATE TYPE rating_enum AS ENUM (
    'ONE', 'TWO', 'THREE', 'FOUR', 'FIVE'
)
```

In the following examples, I use PostgreSQL's enum type. But you can use the same approach to support similar types supported by other DBMS.

Create a DB-Specific Enum Type

Unfortunately, you can't use Hibernate's default mapping to map your Java enum to a PostgreSQL enum. As explained earlier, Hibernate maps the enum values to an *int* or a *String*. But PostgreSQL expects you to set the value as an *Object*.

If you want to map your enum to PostgreSQL's enum type, you need to implement a custom mapping by extending Hibernate's *EnumType*.

```
public class EnumTypePostgreSql extends EnumType {

    @Override
    public void nullSafeSet(PreparedStatement st, Object value, int index,
            SharedSessionContractImplementor session)
            throws HibernateException, SQLException {
```

```
        if(value == null) {
            st.setNull( index, Types.OTHER );
        }
        else {
            st.setObject( index, value.toString(), Types.OTHER );
        }
    }


}
```

And to make it even better, you can use your custom type to map any Java enum to a PostgreSQL enum. So, you can use this type for all enums in your project, and you can even add it to one of your internal libraries.

Use Your Custom Type

You can use the custom type in your entity mappings in 2 ways.

You can reference the class of your enum type in a *@Type* annotation on your entity attribute. This is a good approach if you only use the type on one entity attribute.

```
@Entity
public class Review {

    @Id
    @GeneratedValue
    private Long id;

    private String message;

    @Enumerated(EnumType.STRING)
    @Type(type = "org.thoughts.on.java.model.EnumTypePostgreSql")
    private Rating rating;

    ...
}
```

When you use this mapping, Hibernate uses the *EnumTypePostgreSql* to map the *Rating* value to a PostgreSQL-specific enum type.

If you use the type to map multiple entity attributes, you should register your type using a *@TypeDef* annotation. You can either add the annotation to one of your entities or put it into a *package-info.java* file.

```
@org.hibernate.annotations.TypeDef(name = "enum_postgressql",
                                   typeClass = EnumTypePostgreSql.class)

package org.thoughts.on.java.model;
```

After you've done that, you can reference the type by its logical name in your entity mapping.

```
@Entity
public class Review {

    @Id
    @GeneratedValue
    private Long id;

    private String message;

    @Enumerated(EnumType.STRING)
    @Type(type = "enum_postgressql")
    private Rating rating;

    ...
}
```