8 Ways to use the features of your database

1. Read-only Views

Views are another database feature you can easily use with Hibernate. You can map a view in the same way as any database table. As long as you follow the default naming strategy, you just need a class with an *@Entity* annotation and an attribute for each database column.

You can map read-only view in the same way. You just need an additional annotation to tell Hibernate that it should ignore the entity for all write operations. You can do that with Hibernate's *@Immutable* annotation.

```
@Entity

@Immutable

public class BookView { … }
```

2. Call Database Functions

Calling a database function to perform simple operations, like counting the number of characters in a String, is a standard feature in SQL. You can do the same with JPA and Hibernate.

Call a standard function

JPA and Hibernate support the following set of standard functions which you can use in a JPQL query. The Criteria API supports the same functions and provides one or more methods for each of them.

Function	Description
upper(String s)	Transforms *String s* to upper case
lower(String s)	Transforms *String s* to lower case
current_date()	Returns the current date of the database
current_time()	Returns the current time of the database

Function	Description
current_timestamp()	Returns a timestamp of the current date and time of the database
substring(String s, int offset, int length)	Returns a substring of the given *String s*
trim(String s)	Removes leading and trailing whitespaces from the given *String s*
length(String s)	Returns the length of the given *String s*
locate(String search, String s, int offset)	Returns the position of the *String search* in *s*. The search starts at the position *offset*
abs(Numeric n)	Returns the absolute value of the given number
sqrt(Numeric n)	Returns the square root of the given number
mod(Numeric dividend, Numeric divisor)	Returns the remainder of a division

You can use these functions in the *SELECT* and *WHERE* clause of your query. You can see a simple example in the following code snippet.

```
Query q = em.createQuery("SELECT a, size(a.books) "
                        + "FROM Author a GROUP BY a.id");
```

Call an unsupported function

Most databases support a lot more functions than the ones directly supported by Hibernate or JPA. But don't worry, you can call them anyways.

Since JPA 2.1, you can use the function *function* to call any function supported by your database. You just need to provide the name of the database function as the first parameter followed by the arguments you want to provide to the function call.

I use the function *function* in the following code snippet, to call the user-defined function *calculate* with the *price* of the book and a bind parameter as arguments.

```
TypedQuery<Book> q = em.createQuery(
        "SELECT b FROM Book b WHERE :double2 > "
        "function('calculate', b.price, :double1)"
        , Book.class);
```

Hibernate uses the parameters provided to the function function to call the calculate function in the SQL statement. If you want to learn more about JPA's and Hibernate's support for custom database function calls, take a look at How to call custom database functions with JPA and Hibernate.

3. Stored Procedures

Stored procedures provide another option to perform logic within your database. That can be beneficial, if you need to share logic between multiple applications that use the same database or if you're looking for the most efficient way to implement data-heavy operations.

As you can see in the following code snippets, the annotation-based definition of a stored procedure call isn't complicated. In the first step, you define the stored procedure call with a *@NamedStoredProcedure* annotation by providing the name of the stored procedure and its input and output parameters.

```
@NamedStoredProcedureQuery(
        name = "calculate",
        procedureName = "calculate",
        parameters = {
                @StoredProcedureParameter(
                        mode = ParameterMode.IN,
                        type = Double.class, name = "x"),
                @StoredProcedureParameter(
                        mode = ParameterMode.IN,
                        type = Double.class, name = "y"),
                @StoredProcedureParameter(
                        mode = ParameterMode.OUT,
                        type = Double.class, name = "sum")
        }
)
```

You can then use the *@NamedStoredProcedureQuery* in a similar way as you call a named query. You just have to call the *createNamedStoredProcedureQuery* method of the *EntityManager* with the name of your *@NamedStoredProcedureQuery* to instantiate it. Then you can set the input parameters, execute the query and read the output parameter.

```
StoredProcedureQuery query = this.em
        .createNamedStoredProcedureQuery("calculate");
query.setParameter("x", 1.23d);
query.setParameter("y", 4.56d);
query.execute();
Double sum =
        (Double) query.getOutputParameterValue("sum");
```

4. Database columns with generated values

Another often used feature of relational databases are triggers that initialize or update certain database columns. You can use them, for example, to automatically persist the timestamp of the last update. While you could also do that with Hibernate, most database administrators prefer to handle that on a database level.

But this approach has a drawback. Hibernate has to perform an additional query to retrieve the generated values from the database. That slows down your application and Hibernate doesn't execute the extra query by default.

You need to annotate the attributes that map a database column with a generated value with Hibernate's *@Generated(GenerationTime value)* annotation. The *GenerationTime* annotation tells Hibernate when it has to check for a new value. It can either do that *NEVER*, after each *INSERT* or after each INSERT and UPDATE (*GenerationTime.ALWAYS*) operation.

The following code snippet shows an example of such a mapping and of the SQL statements Hibernate has to perform.

```
@Entity
public class Author {

        @Column
        @Generated(GenerationTime.ALWAYS)
        private LocalDateTime lastUpdate;

        …
}
```

5. Map SQL expressions

Your domain and table model don't need to be identical. You can also map the result of an SQL expression to a read-only attribute of your domain model.

You can do that with Hibernate's *@Formula* annotation. It allows you to specify an SQL expression which Hibernate executes when it reads the entity from the database.

I use it in the following example to calculate the *age* of an *Author* based on her date of birth.

```
@Entity
public class Author {

        @Column
        private LocalDate dateOfBirth;

        @Formula(value = “date_part(‘year’, age(dateOfBirth))”)
        private int age;

        …
}
```

6. Sequences

Database sequences are often used to generate unique primary key values. Hibernate and JPA support different options to generate primary key values and database sequences are, of course, one of them.

If you want to use Hibernate’s default sequence, you just need to annotate your primary key attribute with *@GeneratedValue* and set the strategy to *GenerationType.SEQUENCE*.

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE)
@Column(name = "id", updatable = false, nullable = false)
private Long id;
```

You can also use a custom database sequence when you add a *@SequenceGenerator* annotation. It allows you to define the name and database schema of your sequence and the allocation size Hibernate shall use to retrieve primary key values.

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator = "book_generator")
@SequenceGenerator(name="book_generator",
        sequenceName = "book_seq", allocationSize=50)
@Column(name = "id", updatable = false, nullable = false)
```

7. Autoincremented Database Columns

Autoincremented columns provide another option to generate unique primary key values. The database automatically increments the value of this column for each new record.

The mapping of such a column is similar to the one I showed in the previous example. You just need to tell Hibernate to use a different strategy to generate the primary key values. The *GenerationType.IDENTIFIER* tells Hibernate that the database provides the primary key value.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id", updatable = false, nullable = false)
private Long id;
```

8. Custom and Database-Specific Datatypes

Most databases support a set of custom data types, like PostgreSQL's *JSONB*. JPA and Hibernate don't support them. But that doesn't mean that you can't use them. You just have to implement the mapping yourself.

With Hibernate's *UserType* interface, that is not as complicated as it sounds. Let's have a quick look at the most important steps. If you want to dive deeper into this topic, please take a look at my post How to use PostgreSQL's JSONB data type with Hibernate.

Hibernate's *UserType* interface allows you to define the mapping between any Java type and any supported JDBC data type. That requires the implementation of several methods. The 2 most important ones are *nullSafeGet* and *nullSafeSet*. They implement the mapping from the JDBC to the Java type and vice versa.

The following code snippet shows the implementation of these methods for a *UserType* which maps a Java class to a *JSONB* database column.

```
@Override
public Object nullSafeGet(final ResultSet rs,
        final String[] names, final SessionImplementor session,
        final Object owner)
                throws HibernateException, SQLException {

        final String cellContent = rs.getString(names[0]);
        if (cellContent == null) {
                return null;
        }
        try {
                final ObjectMapper mapper = new ObjectMapper();
                return mapper.readValue(
                        cellContent.getBytes("UTF-8"),
                        returnedClass());
        } catch (final Exception ex) {
                throw new RuntimeException(
                        "Failed to convert String to Invoice: " +
                        ex.getMessage(), ex);
        }
}
```

```
@Override
public void nullSafeSet(final PreparedStatement ps,
        final Object value, final int idx,
        final SessionImplementor session)
                throws HibernateException, SQLException {

        if (value == null) {
                ps.setNull(idx, Types.OTHER);
                return;
        }
        try {
                final ObjectMapper mapper = new ObjectMapper();
                final StringWriter w = new StringWriter();
                mapper.writeValue(w, value);
                w.flush();
                ps.setObject(idx, w.toString(), Types.OTHER);
        } catch (final Exception ex) {
                throw new RuntimeException(
                        "Failed to convert Invoice to String: " +
                        ex.getMessage(), ex);
        }
}
```

After you implemented your own *UserType*, you need to register it. You can do that with a *@TypeDef* annotation which you should add to the *package-info.java file*.

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


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

If the Hibernate dialect doesn't already support the column type, as it's the case for the *JSONB* type, you also need to extend the dialect. As you can see in the following code snippet, this requires only a few lines of code.

```
public class MyPostgreSQL94Dialect extends
PostgreSQL94Dialect {


        public MyPostgreSQL94Dialect() {
                this.registerColumnType(Types.JAVA_OBJECT,
                                                    "jsonb");
        }
}
```