Hibernate Tips

More than 70 solutions to common Hibernate problems

Thorben Janssen

HIBERNATE TIPS

MORE THAN 70 SOLUTIONS TO COMMON HIBERNATE PROBLEMS

Thorben Janssen
Hohe Str. 34
01187 Dresden
Germany
http://www.thoughts-on-java.org

ISBN: 978-1544869179

Copy Editor: Nermina Miller
Cover: ebokks, Hildesheim
Cover Image: BillionPhotos.com – fotolia.com

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is

sold without warranty, either express or implied. The author will not be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

FOREWORD

Undoubtedly, Hibernate ORM and JPA have a steep learning curve. You develop a quick prototype, add a few annotations to your Java classes and everything just works — things seem easy. But, as you try to tackle more complex mappings or resolve performance problems, you quickly realize that you need a deeper understanding of Hibernate to implement a complete and efficient database access layer.

How do you obtain this deeper understanding? The Hibernate documentation is useful, and we always suggest that users read it all. But, that is a daunting task because the documentation contains a lot of content. Also, the structure of the documentation is to describe individual parts of the solutions. However, object/relational mapping is a very complex concept --- it is simply not feasible for a manual to cover all combinations that are often needed to solve real-world problems and implement real-world use cases.

For example, you might not remember the combination of annotations needed to define a specific mapping idea, or you're just wondering how to implement a specific use case. You need a recipe or a quick tip on how to implement the task you're currently working on. For such cases, users have many options to find solutions including Hibernate's blogs, its user forums, its IRC channels, its HipChat rooms, StackOverflow, and so on. Another great resource are the

numerous books on using Hibernate. Additionally a number of blogs exist, dedicated to using Hibernate by community experts --- long-time Hibernate power users. Many of these community expert blogs focus on showing how to use Hibernate's existing features and annotations to implement specific use cases or how to research solving performance problems.

Thorben has been part of this community expert group for a long time, helping Hibernate users via his blog posts, articles, and various forums. And now he has written a book. And, as always, Thorben has a lot of great Hibernate insight to share. This is the first book on Hibernate I have seen that takes an FAQ-style approach, which is an unusual structure. Other books on Hibernate, as well as the Hibernate documentation itself, take the same basic approach to teaching --- they explain the individual pieces in detail, sequentially. While this is valuable (and I'd argue critical) knowledge, it is often hard for new users to apply this sequential, segmented knowledge to resolve more complex topics.

The FAQ approach makes it easier for users to find help on common higher-level concepts and topics. Both forms of knowledge are useful in learning Hibernate. Together with the other listed resources, this book will be a great addition to every developer's Hibernate toolbox.

Steve Ebersole
Lead Developer - Hibernate ORM
Principal Software Engineer - Red Hat, Inc.

PREFACE

Hibernate is one of the most popular Java Persistence API (JPA) implementations and also one of the most popular Java Object Relational Mapping (ORM) frameworks in general. It helps you to map the classes of your domain model to database tables and automatically generate SQL statements to update the database on object state transitions. That is a complex task, but Hibernate makes it look easy. You just annotate your domain classes, and Hibernate takes care of the rest. Or, it at least seems like that in the beginning.

When you've used Hibernate for a while, you begin to recognize that you need to do more than just add an `@Entity` annotation to your domain model classes. Real-world applications often require advanced mappings, complex queries, custom data types, and caching.

Hibernate can do all of that. You just have to know which annotations and APIs to use. The acute need for this knowledge prompted me to write the Hibernate Tips series on my Thoughts on Java blog in 2016. In this book, you'll find more than 35 exclusive tips and the most popular tips from the blog.

What you get in this book

More than 70 Hibernate tips show you how to solve different problems with Hibernate. Each of these tips consists of one or more code samples and an easy-to-follow procedure. You can also download an example project with executable test cases for each Hibernate tip. I recommend downloading this project before you start reading the book so that you can try each Hibernate tip when you read it.

To help you find the tip for your development task, I grouped them into the following chapters:

- I show you how to bootstrap Hibernate in different execution environments in the Setting up Hibernate chapter.
- In the Basic Mappings chapter, I introduce you to basic attribute mappings that allow you to use Hibernate's standard mappings to map an entity to a database table.
- The tips in the Advanced Mappings chapter show you some of Hibernate's advanced features and how you can use them for things like defining custom mappings for unsupported data types, mapping of read-only database views, defining derived primary keys, and mapping of inheritance hierarchies.
- Hibernate implements the JPA specification, but it also provides several proprietary features. I show you some of them in the Hibernate Specific Queries and Mappings chapter.
- Java 8 introduced several new APIs and programming concepts. Since version 5, you can use them with Hibernate. I show you a few examples in the Java 8 chapter.

- Logging is an important topic that gets ignored in a lot of projects. You should always make sure that Hibernate logs useful information during development and doesn't slow down your application in production. I give you several configuration tips in the Logging chapter.
- The tips in the JPQL chapter show you how to use JPA's query language to read records from the database and how you can use it to update or delete multiple entities at once.
- If your queries are too complex for JPQL, take a look at the Native SQL Queries chapter, which shows how to perform native SQL queries with Hibernate.
- The Criteria API provides another option to create database queries. It is especially useful if you need to create queries programmatically. I show you several examples using this API in the Create queries programmatically with the Criteria API chapter.
- In the Stored Procedures chapter, I explain how you can use the `@NamedStoredProcedureQuery` annotation and the `StoredProcedureQuery` interface to execute stored procedures in your database.
- Caching can be an effective approach to improve the performance of your application. I show you how to activate and use Hibernate's second-level and query cache in the Caching chapter.

How to get the example project

I use a lot of code samples in this book to show you how to solve a specific problem with Hibernate. You can download an example project with all code samples and executable test cases at http://www.hibernate-tips.com/download-examples.

Who this book is for

This book is for developers who are already working with Hibernate and who are looking for solutions for their current development tasks or problems. The tips are designed as self-contained recipes that provide specific solutions and can be accessed as needed. Most tips contain links to related tips that you can follow if you want to dive deeper into a topic or need a slightly different solution. There is no need to read the tips in a specific order. Feel free to read the book from cover to cover or just pick the tips that help you in your current project.

To get the most out of this book, you should already be familiar with the general concepts of JPA and Hibernate. You're in the right place if you are looking for tips on how to use Hibernate to implement your business requirements. I don't explain Hibernate's general concepts, and therefore this book is not intended for beginners. But, if you're already familiar with ORM frameworks and like to learn by doing, you may find this example-based approach helpful.

SETTING UP HIBERNATE

You can use Hibernate in several different environments. You can use it as a JPA implementation in a Java SE or Java EE environment, as a proprietary persistence framework in Java SE or as a persistence provider in Spring. The core Hibernate features that I explain in the Hibernate Tips in this book, are available in all these environments. The only differences are features that other frameworks in your environment provide on top of Hibernate and the bootstrapping mechanism.

You can find examples for the different bootstrapping approaches in the following Hibernate tips:

- How to bootstrap Hibernate in a Java SE environment
- How to bootstrap Hibernate in a Java EE environment
- How to use Hibernate's native bootstrapping API
- How to bootstrap Hibernate with Spring Boot
- How to access Hibernate APIs from JPA
- How to automatically add Metamodel classes to your project

How to bootstrap Hibernate in a Java SE environment

Problem

I want to use Hibernate as my JPA provider in a Java SE environment. How do I bootstrap Hibernate?

Solution

Before you can bootstrap Hibernate in your Java SE application, you need to add the required dependencies to your classpath. I'm using Hibernate 5.2.8.Final for the examples of this book, and the `hibernate-core.jar` file is the only required Hibernate dependency. The JPA jar-file is included as a transitive dependency of `hibernate-core`.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.8.Final</version>
</dependency>
```

You also need to add a database-specific JDBC driver to the classpath of your application. Please check your database documentation for more information.

After you've added the required dependencies, you can bootstrap Hibernate as it is defined in the JPA specification. You need to add a `persistence.xml` file to the `META-INF` directory of your application. The following code snippet shows a simple example of a `persistence.xml` file. It configures a persistence-unit with the name `my-persistence-unit`. It also tells Hibernate to use the `PostgreSQLDialect` and to connect to a PostgreSQL

database on localhost. Your configuration might differ if you use a different database or a connection pool.

```
<persistence>
    <persistence-unit name="my-persistence-unit">
        <description>Hibernate Tips</description>
        <provider>
            org.hibernate.jpa.HibernatePersistenceProvider
        </provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>

        <properties>
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.PostgreSQLDialect" />

            <property name="javax.persistence.jdbc.driver"
                value="org.postgresql.Driver" />
            <property name="javax.persistence.jdbc.url"
                value="jdbc:postgresql://localhost:5432/recipes" />
            <property name="javax.persistence.jdbc.user"
                value="postgres" />
            <property name="javax.persistence.jdbc.password"
                value="postgres" />
        </properties>
    </persistence-unit>
</persistence>
```

You can then call the `createEntityManagerFactory` of the `Persistence` class to create an `EntityManagerFactory` for the `persistence-unit` you configured in your `persistence.xml` file. The `EntityManagerFactory` provides a method to get an `EntityManager`, which I use in most examples in this book. That's all you need to do to bootstrap Hibernate in your application.

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("my-persistence-unit");
EntityManager em = emf.createEntityManager();
```

Source Code

You can find a project with executable test cases for this Hibernate Tip in the `JPABootstrapping` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

JPA also defines a bootstrapping approach for Java EE environments. I explain it in How to bootstrap Hibernate in a Java EE environment.

You can also use Hibernate's proprietary bootstrapping API, which gives you access to proprietary configuration features. I show you how to do that in How to use Hibernate's native bootstrapping API.

If you want to use Hibernate with Spring Boot, take a look at How to bootstrap Hibernate with Spring Boot.

How to bootstrap Hibernate in a Java EE environment

Problem

I want to use Hibernate as my JPA provider in a Java EE environment. How do I bootstrap Hibernate?

Solution

Bootstrapping Hibernate in a Java EE environment is pretty simple. You need to make sure that Hibernate is set up in your Java EE application server. That's the case if you're using a JBoss Wildfly or JBoss EAP server. Please check your application server documentation if you're using a different server. Your server might already use Hibernate as the JPA implementation or you need to decide if you want to replace the existing JPA implementation.

You can then bootstrap Hibernate as it is defined in the JPA specification. You just need to add a `persistence.xml` file to the `META-INF` directory of a deployment unit. The following code snippet shows a simple example of a `persistence.xml` file that defines the persistence-unit `my-persistence-unit`. It tells Hibernate to use the `PostgreSQLDialect` and to connect to a PostgreSQL database on localhost. Your configuration might differ if you use a connection pool provided by your application server.

```
<persistence>
    <persistence-unit name="my-persistence-unit">
        <description>Hibernate Tips</description>
        <provider>
            org.hibernate.jpa.HibernatePersistenceProvider
        </provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
```

```
        <properties>
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.PostgreSQLDialect" />

            <property name="javax.persistence.jdbc.driver"
                value="org.postgresql.Driver" />
            <property name="javax.persistence.jdbc.url"
                value="jdbc:postgresql://localhost:5432/recipes" />
            <property name="javax.persistence.jdbc.user"
                value="postgres" />
            <property name="javax.persistence.jdbc.password"
                value="postgres" />
        </properties>
    </persistence-unit>
</persistence>
```

The container creates an `EntityManagerFactory` for each `persistence-unit` defined in the configuration. It also enables you to inject an `EntityManagerFactory` or an `EntityManager` when you need it.

```
@PersistenceUnit
private EntityManagerFactory emf;

@PersistenceUnit
private EntityManager em;
```

Learn More

JPA and Hibernate also provide two approaches to bootstrap Hibernate in a Java SE environment. I explain them in How to bootstrap Hibernate in a Java SE environment and How to use Hibernate's native bootstrapping API.

If you want to use Hibernate with Spring Boot, take a look at How to bootstrap Hibernate with Spring Boot.

How to use Hibernate's native bootstrapping API

Problem

I need more control over Hibernate's internal configuration. How do I use its native bootstrapping API?

Solution

Hibernate's native bootstrapping API is very flexible, which makes it more complicated to use but also more powerful than the JPA bootstrapping API. If you don't need this flexibility, I recommend using the JPA API.

Before you can start the bootstrapping process, you need to add the required dependencies to your classpath. I'm using Hibernate 5.2.8.Final for the examples of this book, and the `hibernate-core.jar` file is the only required Hibernate dependency. It also includes the JPA jar-file as a transitive dependency.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.8.Final</version>
</dependency>
```

You also need to add a database-specific JDBC driver to the classpath of your application. Please check your database documentation for more information.

As soon as you add the required dependencies, you can implement the bootstrapping process. You need to create a `StandardServiceRegistry`,

build a `Metadata` object, and use that object to instantiate a `SessionFactory`.

Hibernate uses two service registries --- the `BootstrapServiceRegistry` and the `StandardServiceRegistry`. The default `BootstrapServiceRegistry` provides a good solution for most applications, so I skip the programmatic definition in this example.

However, you need to configure the `StandardServiceRegistry`. In this example, I do that using a `hibernate.cfg.xml` file, which makes the implementation easy and allows you to change the configuration without changing the source code. Hibernate loads the configuration file automatically from the classpath when you call the `configure` method on the `StandardServiceRegistryBuilder`. You can then adapt the configuration programmatically before you call the `build` method to get a `ServiceRegistry`.

```
ServiceRegistry standardRegistry =
        new StandardServiceRegistryBuilder()
                .configure()
                .build();
```

The following code snippet shows an example of a `hibernate.cfg.xml` configuration file. It tells Hibernate to use the `PostgreSQLDialect` and to connect to a PostgreSQL database on localhost. It also tells Hibernate to generate the database tables based on the entity mappings. Your configuration may differ if you use a different database or a connection pool.

> Generating your database tables based on entity mappings is not recommended for production. You should use SQL scripts instead so that you are in control of your database model and can optimize it for your requirements.

```
<hibernate-configuration>
    <session-factory>
        <property name="dialect">
            org.hibernate.dialect.PostgreSQLDialect
        </property>

        <property name="connection.driver_class">
            org.postgresql.Driver
        </property>
        <property name="connection.url">
            jdbc:postgresql://localhost:5432/recipes
        </property>
        <property name="connection.username">postgres</property>
        <property name="connection.password">postgres</property>
        <property name="connection.pool_size">1</property>

        <property name="hbm2ddl.auto">create</property>
    </session-factory>
</hibernate-configuration>
```

After you instantiate a configured `ServiceRegistry`, you need to create a `Metadata` representation of your domain model. You can do that based on the configuration files `hbm.xml` and `orm.xml` or annotated entity classes. I use annotated classes in the following code snippet. I first use the `ServiceRegistry`, which I created in the previous step to instantiate a new `MetadataSources` object. Then I add my annotated entity classes and call the `buildMetadata` to create the `Metadata` representation. In this example, I use only the `Author` entity. After that, I call the `buildSessionFactory` method on the `Metadata` object to instantiate a `SessionFactory`.

```
SessionFactory sessionFactory =
        new MetadataSources(standardRegistry)
                .addAnnotatedClass(Author.class)
                .buildMetadata()
                .buildSessionFactory();
Session session = sessionFactory.openSession();
```

That is all you need to do to create a basic Hibernate setup with its native API. You can now use the `SessionFactory` to open a new `Session` and use it to read or persist entities.

```
Author a = new Author();
a.setFirstName("Thorben");
a.setLastName("Janssen");
session.persist(a);
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `HibernateBootstrapping` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

The bootstrapping API defined by the JPA standard is easier to use but not as flexible. I explain it in more detail in:

- How to bootstrap Hibernate in a Java SE environment
- How to bootstrap Hibernate in a Java EE environment

You can also use Hibernate with Spring Boot. I explain the required bootstrapping process in How to bootstrap Hibernate with Spring Boot.

How to bootstrap Hibernate with Spring Boot

Problem

How do I use Hibernate in my Spring Boot application?

Solution

Spring Boot makes it extremely easy to bootstrap Hibernate. You just need to add the Spring Boot JPA starter to your classpath, and Spring Boot handles the bootstrapping for you.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

You also need to add a database-specific JDBC driver to the classpath of your application. Please check your database documentation for more information.

You define your data source with a few properties in the `application.properties` file. The following configuration example defines a data source that connects to a PostgreSQL database on localhost.

```
spring.datasource.url = jdbc:postgresql://localhost:5432/recipes
spring.datasource.username = postgres
spring.datasource.password = postgres
```

If you add an H2, HSQL, or Derby database on the classpath, you can safely omit the configuration, and Spring Boot starts and connects to an in-memory database. You can also add multiple JDBC drivers and an in-memory database to your classpath and use different configurations for different target environments.

That’s all you need to do bootstrap Hibernate in a Spring Boot application. You can now use the `@Autowired` annotation to inject an `EntityManager`.

```
@Autowired
private EntityManager em;
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `SpringBootBootstrapping` module of the example project. If you haven’t already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

JPA defines a bootstrapping approach for Java SE and Java EE environments. I explain it in:

- How to bootstrap Hibernate in a Java SE environment
- How to bootstrap Hibernate in a Java EE environment

You can also use Hibernate’s proprietary bootstrapping API, which gives you access to proprietary configuration features. I show you how to do that in How to use Hibernate’s native bootstrapping API.

How to access Hibernate APIs from JPA

Problem

I'm using Hibernate via the `EntityManager` API. Is there a way to access the proprietary Hibernate `Session` and `SessionFactory`?

Solution

Since version 2.0, JPA provides easy access to the APIs of the underlying implementations. `EntityManager` and `EntityManagerFactory` provide a n `unwrap` method, which returns the corresponding classes of the JPA implementation. In Hibernate's case, `Session` and `SessionFactory` give you full access to proprietary Hibernate features, such as the support for `Streams` and `Optional`.

The following code snippet shows you how to get the Hibernate `Session` from `EntityManager`. You just need to call the `unwrap` method on `EntityManager` and provide the `Session` class as a parameter.

```
Session session = em.unwrap(Session.class);
```

As you can see in the next code snippet, you can get Hibernate's `SessionFactory` in a similar way. You first get `EntityMangerFactory` f r o m `EntityManager` and then call the unwrap method with the `SessionFactory` class.

```
SessionFactory sessionFactory = em.getEntityManagerFactory()
                                  .unwrap(SessionFactory.class);
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `AccessHibernateApi` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

You also get direct access to Hibernate's `Session` and `SessionFactory` classes, if you use its native bootstrapping API. For more information, see How to use Hibernate's native bootstrapping API.

How to automatically add Metamodel classes to your project

Problem

I use Hibernate's Static Metamodel Generator to generate the JPA Metamodel. These classes are generated to a different directory, which isn't used as a source folder. Is there a way to automatically register this folder as a source folder?

Solution

During my research, I learned from Frits Walraven that there is a Maven plugin that can do exactly that. Special thanks to Frits, who also reviewed this book.

The only thing you need to do is to add the following Maven plugin to your build configuration. It registers a list of directories as additional source folders. I use it in the parent `pom.xml` file of my project to add the directory, to which the JPA Metamodel classes get generated (`target/generated-sources/annotations`), as a source folder.

```
<project>
    ...

    <build>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>build-helper-maven-plugin</artifactId>
                <version>3.0.0</version>
                <executions>
                    <execution>
                        <id>add-source</id>
                        <phase>generate-sources</phase>
```

```
                        <goals>
                            <goal>add-source</goal>
                        </goals>
                        <configuration>
                            <sources>
                                <source>
                                target/generated-sources/annotations
                                </source>
                            </sources>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

    ...
</project>
```

Source Code

You can find an example of a complete maven build configuration in the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

The JPA Metamodel provides a type-safe way to reference entity attributes when you create a `CriteriaQuery` or an `EntityGraph`. I explain it in more detail in How to reference entity attributes in a type-safe way.

BASIC MAPPINGS

Hibernate supports a set of standard mappings that allows you to map an entity attribute to a database column easily. The examples in this chapter show you how to use standard mappings for entity attributes, map primary key attributes, generate unique primary key values, and map associations between entities. You can find more advanced mapping definition in the Advanced Mappings chapter.

To learn about standard mappings for entity attributes, see:

- How to define schema and table names
- How to map basic entity attributes to database columns
- How to map a util Date or Calendar to a database column
- How to map an enum to a database column

To learn about the mapping of primary key attributes and generation of unique primary key values, see:

- How to map a simple primary key
- How to use an auto-incremented column to generate primary key values

- How to use a custom database sequence to generate primary key values
- How to use a database table to generate primary key values
- How to use a generated UUID as a primary key

To learn about the mapping of associations between entities, see:

- How to map a bidirectional many-to-one association
- How to map an unidirectional many-to-one association
- How to map an unidirectional one-to-many association
- How to map a bidirectional many-to-many association
- How to map an unidirectional many-to-many association
- How to map a bidirectional one-to-one association
- How to map an unidirectional one-to-one association

How to define schema and table names

Problem

How do I define the name of the database schema and table to which Hibernate maps an entity?

Solution

You can define the schema and table name with the `schema` and `name` attributes of the `javax.persistence.Table` annotation. See a related example in the following code snippet. You just have to add the `@Table` annotation to your entity class and set the name and schema attributes.

```
@Entity
@Table(name = "author", schema = "bookstore")
public class Author {...}
```

When you now use the entity, Hibernate uses the provided schema and table names to create the SQL statements. The following code snippet persists a new `Author` entity and performs a query to get all `Author` entities with the given first name.

```
Author a = new Author();
a.setFirstName("firstName");
a.setLastName("lastName");
em.persist(a);
a = em.createQuery(
    "SELECT a FROM Author a WHERE firstName = ‘firstName'",
    Author.class).getSingleResult();
```

As you can see in the following log output, Hibernate persists the `Author` entity to the `author` table in the bookstore database schema and performs the `SELECT` statement on the same table.

```
06:11:42,481 DEBUG [org.hibernate.SQL] -
    insert
    into
        bookstore.author
        (firstName, lastName, version, id)
    values
        (?, ?, ?, ?)
06:11:42,785 DEBUG [org.hibernate.SQL] -
    select
        author0_.id as id1_0_,
        author0_.firstName as firstNam2_0_,
        author0_.lastName as lastName3_0_,
        author0_.version as version4_0_
    from
        bookstore.author author0_
    where
        author0_.firstName='firstName'
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `DefineTableAndSchemaName` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

How to map basic entity attributes to database columns

Problem

How do I map basic entity attributes to database columns?

Solution

Hibernate doesn't need any additional information to map entity attributes of the following Java types:

- `java.lang.String`
- `char` and `java.lang.Character`
- `boolean` and `java.lang.Boolean`
- `byte` and `java.lang.Byte`
- `short` and `java.lang.Short`
- `int` and `java.lang.Integer`
- `long` and `java.lang.Long`
- `float` and `java.lang.Float`
- `double` and `java.lang.Double`
- `java.math.BigInteger`
- `java.math.BigDecimal`
- `java.sql.Timestamp`
- `java.sql.Time`
- `java.sql.Date`

- `java.util.Calendar`
- `java.util.Date`
- `java.util.Locale`
- `java.util.Timezone`
- `java.net.URL`
- `java.sql.Blob`
- `java.sql.Clob`
- `byte[]` and `java.lang.Byte[]`
- `char[]` and `java.lang.Character[]`
- `java.util.UUID`
- `java.sql.NClob`

Hibernate also supports a default mapping for enumerations which doesn't require any additional annotations. I get into more detail about that mapping and its customization options in How to map an enum to a database column.

Since version 5.0, Hibernate also supports the classes of the Date and Time API as basic types:

- `java.time.Duration`
- `java.time.Instant`
- `java.time.LocalDateTime`
- `java.time.LocalDate`
- `java.time.LocalTime`
- `java.time.OffsetDateTime`

- `java.time.OffsetTime`
- `java.time.ZonedDateTime`

When you don't provide any additional mapping information, Hibernate maps entity attributes of these types to database columns with the same name. It maps the entity attributes `id`, `version`, and `firstName` in the following code snippet to the database columns `id`, `version`, and `firstname`.

If you want to map an entity attribute to a column with a different name, you can annotate it with `@Column` and provide the column name as the `name` attribute. I use this annotation in the following code snippet to map the `lastName` attribute to the database column `lname`.

You can also use the `@Column` annotation to provide additional mapping information that Hibernate can use to generate the database schema and to apply internal optimizations. I don't recommend using Hibernate's schema generation feature to create the final version of your table model, and you shouldn't add any additional annotations for it to your entity mappings. But, you should tell Hibernate if a column is `nullable`, `insertable`, and `updatable` because it can use this information for internal optimizations. I show that in the following example for the primary key attribute `id`. The mapping doesn't allow `null` as a value, and Hibernate doesn't support updates on primary key attributes.

```
@Entity
public class Author {

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

```
    @Version
    private int version;

    private String firstName;

    @Column(name = "lname")
    private String lastName;

    ...
}
```

When you now use the entity, Hibernate uses the default mapping for all columns that are not annotated. For all other columns, Hibernate uses the mapping information provided by the annotation. The following code snippet persists a new `Author` entity and performs a query to get the `Author` entity with a given `id`.

```
Author a = new Author();
a.setFirstName("John");
a.setLastName("Doe");
em.persist(a);

...

a = em.find(Author.class, a.getId());
```

As you can see in the following log output, Hibernate persists the Author entity to the author table and uses the default column names for the entity attributes `id`, `version`, and `firstName`. Hibernate doesn't use the default name for the `lastName` attribute because the `@Column` annotation specifies a mapping to the column name `lname`. And, of course, Hibernate uses the same mappings for the `SELECT` statement.

```
12:03:01,827 DEBUG [org.hibernate.SQL] -
    insert
    into
        Author
        (firstName, lname, version, id)
    values
        (?, ?, ?, ?)

...

12:03:01,849 DEBUG [org.hibernate.SQL] -
    select
        author0_.id as id1_0_0_,
        author0_.firstName as firstNam2_0_0_,
        author0_.lname as lname3_0_0_,
        author0_.version as version4_0_0_
    from
        Author author0_
    where
        author0_.id=?
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `MapBasicAttributes` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

Hibernate can map `java.util.Date` and `java.util.Calendar` classes without any additional annotation. It always maps them as an SQL `TIMESTAMP` with milliseconds. But that's often not the mapping you need for your use case. I show you how to choose a different format in How to map a util Date or Calendar to a database column.

The same applies for enumerations. Hibernate can map them by default, but you might want to customize the way it does it. I show you how to do that in How to map an enum to a database column.

How to map a util Date or Calendar to a database column

Problem

I use a `java.util.Date` to model a date as an entity attribute. But Hibernate maps it to a timestamp with nanoseconds. How do I change the mapping so that Hibernate only stores the years, months, and days?

Solution

The SQL standard supports three different data types to store date and time information. Hibernate can map all of them to a `java.util.Date` or a `java.util.Calendar`. You need to decide which of the following SQL types Hibernate will use:

- `TIMESTAMP`: Persists the date and time with nanoseconds. Hibernate uses this type by default.
- `TIME`: Stores only the time of day without nanoseconds.
- `DATE`: Persists only the date with years, months, and days.

You can define the preferred mapping with the `@Temporal` annotation. As you can see in the following code snippet, the annotation takes a `TemporalType` enum as a value. The enum allows you to select the SQL type (`DATE`, `TIME`, or `TIMESTAMP`) that you want to use.

```
@Entity
public class Author {
```

```
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;

    ...
}
```

As you can see in the following log output, the `dateOfBirth` attribute of the `Author` entity gets mapped to an SQL `DATE` without any time information.

```
07:22:50,453 DEBUG [org.hibernate.SQL] -
    select
        author0_.id as id1_0_0_,
        author0_.dateOfBirth as dateOfBi2_0_0_,
        author0_.firstName as firstNam3_0_0_,
        author0_.lastName as lastName4_0_0_,
        author0_.version as version5_0_0_
    from
        Author author0_
    where
        author0_.id=?
07:22:50,454 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [1] as [BIGINT] - [1]
07:22:50,464 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([dateOfBi2_0_0_] : [DATE]) - [1980-01-01]
...
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `MapUtilDate` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Since Hibernate 5, you can also use the classes of the Java 8 Date and Time API as entity attribute types. The new classes solve a lot of issues with the

`java.util.Date` and provide all information Hibernate needs to map them to the correct JDBC types. I explain the mapping of the Date and Time API classes in more detail in How to map classes of Java 8’s Date and Time API.

How to map an enum to a database column

Problem

How do I map an enum attribute to a database column? Which option should I choose?

Solution

JPA and Hibernate provide two standard options to map an enum to a database column. You can either use the `String` representation or the ordinal value.

Both approaches have their drawbacks:

- The `String` representation is verbose, and renaming an enum value requires that you also update your database.
- The ordinal of an enum value is its position in the enum declaration. This value changes and requires you to update your database when you remove an existing value or do not add new values to the end of the enum declaration.

You have to decide which drawback is the lesser evil for your specific application or use an `AttributeConverter` to define a custom mapping to avoid these issues.

When you use the JPA and Hibernate standard mapping, you can either rely on the default mapping of its ordinal value or specify the mapping approach. You can do that with an `@Enumerated` annotation.

The following examples use the `AuthorStatus` enum. This enum indicates if an author published a book with a publisher, self-published, or is still writing a

book.

```
public enum AuthorStatus {

    PUBLISHED, SELF_PUBLISHED, NOT_PUBLISHED;
}
```

The following example mapping uses the `@Enumerated` annotation to explicitly tell Hibernate to use the ordinal value. If you don't provide an `@Enumerated` annotation or don't set an `EnumType` as its value, Hibernate also uses the ordinal value as the default mapping.

```
@Entity
public class Author {

    @Enumerated(EnumType.ORDINAL)
    private AuthorStatus status;

    ...
}
```

When you use this mapping to persist an `Author` entity with `status PUBLISHED`, Hibernate stores the value `0` in the database.

If you want to store the `String` representation of the enum value in the database, you need to annotate the entity attribute with `@Enumerated` and set `EnumType.STRING` as its value.

```
@Entity
public class Author {

    @Enumerated(EnumType.STRING)
    private AuthorStatus status;
```

```
    ...
}
```

When you now persist the same entity in the database, Hibernate writes the value `PUBLISHED` into the database column `status`.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `MapEnumerations` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

You can use an `AttributeConverter` to define your own mapping and avoid the drawbacks of the JPA standard mapping. I show you how to do that in How to define a custom enum mapping.

How to map a simple primary key

Problem

How do I map a simple primary key of a database table with an entity?

Solution

You can model a simple primary key with an entity attribute that you annotate with an `@Id` annotation. JPA and Hibernate support the following data types as primary keys:

- Primitive Java types and their wrapper types
- `java.lang.String`
- `java.util.Date` and `java.sql.Date`
- `java.math.BigDecimal`
- `java.math.BigInteger`

The following code snippet shows an example of a primary key attribute. It is of type `java.lang.Long`. I annotated it with an `@Id` and a `@Column` annotation. You don't need the `@Column` annotation to define a primary key, but it allows you to provide additional mapping information. In this example, I tell Hibernate that the `id` attribute can't be null and can't be updated. You could also provide the database column name, if you don't want to rely on Hibernate's default mapping.

```
@Entity
public class Author {
```

```
    @Id
    @Column(updatable = false, nullable = false)
    private Long id;

    ...
}
```

That's all you need to do to define a primary key attribute. You can now use it, for example, to read an entity via the `EntityManager` find method.

```
Author a = em.find(Author.class, 1L);
```

When you persist a new `Author` entity, you need to provide a unique primary key value.

```
Author a = new Author();
a.setId(1L);
a.setFirstName("Thorben");
a.setLastName("Janssen");

em.persist(a);
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `PrimaryKey` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

JPA and Hibernate support different strategies to generate unique primary key values. You should use one of these strategies if the primary key value is not provided as user input. I explain them in more detail in:

- How to use a custom database sequence to generate primary key values
- How to use an auto-incremented column to generate primary key values
- How to use a database table to generate primary key values
- How to use a generated UUID as a primary key

How to use an auto-incremented column to generate primary key values

Problem

How do I use an auto-incremented database column to generate primary key values?

Solution

JPA and Hibernate support different strategies to generate primary key values. One of them is the identity strategy that uses an auto-incremented database column.

If you want to use this strategy, you have to annotate the primary key attribute with an `@Id` and a `@GeneratedValue` annotation with `GenerationType.IDENTITY` as the value of the strategy attribute.

The following code snippet shows an example of this annotation.

```
@Entity
public class Author {

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

    ...
}
```

If you persist a new `Author` entity, Hibernate performs the SQL `INSERT` statement immediately. It uses the auto-incremented database column `id` to generate the primary key value and retrieves the value from the database. You can see that in the log file if you activate the logging for SQL statements.

```
Author a = new Author();
a.setFirstName("Thorben");
a.setLastName("Janssen");

log.info("Before persist");
em.persist(a);
log.info("After persist");
```

```
05:35:43,918 INFO  [org.thoughts.on.java.model.TestIdentityStrategy]
- Before persist
05:35:43,975 DEBUG [org.hibernate.SQL] -
    insert
    into
        Author
        (firstName, lastName, version)
    values
        (?, ?, ?)
05:35:43,989 DEBUG [org.hibernate.id.IdentifierGeneratorHelper] -
Natively generated identity: 1
05:35:43,993 INFO  [org.thoughts.on.java.model.TestIdentityStrategy]
- After persist
```

> Databases handle auto-incremented columns very efficiently. However, you need to be aware that Hibernate has to perform the `INSERT` statement immediately to get the primary key value. That prevents Hibernate from using different performance optimization techniques that rely on the delayed execution of database operations.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `PrimaryKeyIdentityStrategy` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

Auto-incremented database columns are only one out of four options to generate primary key values. You should also take a look at:

- How to use a custom database sequence to generate primary key values
- How to use a database table to generate primary key values
- How to use a generated UUID as a primary key

How to use a custom database sequence to generate primary key values

Problem

Hibernate uses its default database sequence to generate primary key values. How do I use my own sequence?

Solution

The JPA specification supports four options to generate primary key values. One of them is the `GenerationType.SEQUENCE`, which uses a database sequence to generate primary key values.

When you want to use a custom database sequence, you have to annotate the primary key attribute with the `@GeneratedValue` annotation and set `GenerationType.SEQUENCE` as the value of the `strategy` attribute. This tells Hibernate to use a database sequence to generate the primary key value. If you don't provide any additional information, Hibernate uses its default sequence, `hibernate_sequence`.

You can configure the name and schema of a custom database sequence using a `@SequenceGenerator` annotation. The following code snippet shows an example of such a mapping. The `@GeneratedValue` annotation references a custom `generator` with the name `author_generator`. This generator gets defined by the `@SequenceGenerator` annotation, which tells Hibernate to use the `author_seq` database sequence.

```
@Entity
public class Author {

        @Id
        @GeneratedValue(
            strategy = GenerationType.SEQUENCE,
            generator = "author_generator")
        @SequenceGenerator(
            name="author_generator",
            sequenceName = "author_seq")
        @Column(name = "id", updatable = false, nullable = false)
        private Long id;

        ...

}
```

When you persist a new `Author` entity, Hibernate selects a new primary key value from the database sequence `author_seq` before it executes the SQL `INSERT` statement. You can see these statements in the log file, if you activate the logging for SQL statements.

```
Author a = new Author();
a.setFirstName("Thorben");
a.setLastName("Janssen");

em.persist(a);
```

```
12:17:47,432 DEBUG [org.hibernate.SQL] -
    select
        nextval ('author_seq')
12:17:47,459 DEBUG [org.hibernate.SQL] -
    insert
    into
        Author
        (firstName, lastName, version, id)
    values
        (?, ?, ?, ?)
```

```
12:17:47,462 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [1] as [VARCHAR] - [firstName]
12:17:47,462 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [2] as [VARCHAR] - [lastName]
12:17:47,463 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [3] as [INTEGER] - [0]
12:17:47,464 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [4] as [BIGINT] - [1]
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `CustomSequence` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

Sequences are only one out of four options to generate primary key values. You should also have a look at:

- How to use an auto-incremented column to generate primary key values
- How to use a database table to generate primary key values
- How to use a generated UUID as a primary key

How to use a database table to generate primary key values

Problem

How do I generate primary key values if my database doesn't support sequences or auto-incremented columns?

Solution

JPA and Hibernate support different strategies to generate primary key values. One of them is the table strategy, which uses a database table to simulate a sequence. This strategy provides a good solution if your database doesn't support sequences and auto-incremented database columns.

> Almost all modern databases support sequences or auto-incremented columns that generate primary key values more efficiently than the table strategy described in this tip. Please check if your database supports one of these strategies before you decide to use the table strategy.

If you want to use this strategy, you have to annotate the primary key attribute with an `@Id` and a `@GeneratedValue` annotation with `GenerationType.TABLE` as the value of the strategy attribute.

```
@Entity
public class Author {

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

```
    ...
}
```

When you persist a new `Author` entity, Hibernate selects the next primary key value from the `hibernate_sequences` table and updates it afterwards. These two statements create an overhead and lock the `hibernate_sequence` table until you commit the transaction. That makes the `TABLE` strategy slower than the `SEQUENCE` or `IDENTITY` strategy.

Hibernate then uses the retrieved primary key value to insert the new `Author` entity into the `Author` table. You can see that in the log file if you activate the logging for SQL statements.

```
Author a = new Author();
a.setFirstName("Thorben");
a.setLastName("Janssen");

em.persist(a);
```

```
12:20:53,967 DEBUG [org.hibernate.SQL] -
    select
        tbl.next_val
    from
        hibernate_sequences tbl
    where
        tbl.sequence_name=? for update
            of tbl
12:20:53,972 DEBUG [org.hibernate.SQL] -
    update
        hibernate_sequences
    set
        next_val=?
    where
        next_val=?
        and sequence_name=?
```

```
12:20:54,001 DEBUG [org.hibernate.SQL] -
    insert
    into
        Author
        (firstName, lastName, version, id)
    values
        (?, ?, ?, ?)
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `PrimaryKeyTableStrategy` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

Auto-incremented database columns are only one of multiple options to generate primary key values. You should also take a look at:

- How to use a custom database sequence to generate primary key values
- How to use an auto-incremented column to generate primary key values
- How to use a generated UUID as a primary key

How to use a generated UUID as a primary key

Problem

I want to use a UUID as a primary key. How do I map them with Hibernate? Can Hibernate generate UUID values for new entities?

Solution

Hibernate provides proprietary support for attributes of type `java.util.UUID` as primary keys and offers two generators to create UUID values. The generators support the standards IETF RFC 4122 version 4 and IETF RFC 4122 version 1.

IETF RFC 4122 version 4 --- Random number-based UUID

By default, Hibernate uses a random number-based generation strategy. As always, you don't have to do much to get the default behavior. You just need to add a `@GeneratedValue` annotation to a primary key attribute of type `java.util.UUID`.

You can see an example of it in the following code snippet.

```
@Entity
public class Author {

    @Id
    @GeneratedValue
    @Column(name = "id", updatable = false, nullable = false)
    private UUID id;

    ...
}
```

When you now persist a new `Author` entity, Hibernate generates a UUID before writing the new record to the database.

```
Author a = new Author();
a.setFirstName("Thorben");
a.setLastName("Janssen");

em.persist(a);
```

```
12:25:31,071 DEBUG
[org.hibernate.event.internal.AbstractSaveEventListener] - Generated
identifier: 35a18e65-97b9-48fd-a547-56f81e157253, using strategy:
org.hibernate.id.UUIDGenerator
12:25:31,113 DEBUG [org.hibernate.SQL] -
    insert
    into
        Author
        (firstName, lastName, version, id)
    values
        (?, ?, ?, ?)
12:25:31,117 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [1] as [VARCHAR] - [Thorben]
12:25:31,118 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [2] as [VARCHAR] - [Janssen]
12:25:31,119 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [3] as [INTEGER] - [0]
12:25:31,120 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [4] as [OTHER] - [35a18e65-97b9-48fd-a547-
56f81e157253]
```

IETF RFC 4122 version 1 --- IP- and timestamp-based UUID

Hibernate can also generate a UUID based on the IP and timestamp as defined by IETF RFC 4122 version 1. But the configuration of it requires an additional annotation.

You need to annotate a primary key attribute of type `java.util.UUID` with a `@GeneratedValue` annotation that references a custom ID generator by its name. The ID generator defines the strategy Hibernate uses to generate the primary key value. You can define it with Hibernate's `@GenericGenerator` annotations. It requires the `name` of the generator, the name of the class that implements the generator `strategy`, and an additional `@Parameter` annotation. In this case, the generator is implemented by the `org.hibernate.id.UUIDGenerator` class. That is the same generator class that Hibernate uses by default to generate UUIDs. You need to tell it to use a different generation strategy. You can do that using the `@Parameter` annotation. You need to set the name to `uuid_gen_strategy_class` and provide the fully qualified class name of the `CustomVersionOneStrategy` class as the value.

```
@Entity
public class Book {

    @Id
    @GeneratedValue(generator = "UUID")
    @GenericGenerator(
        name = "UUID",
        strategy = "org.hibernate.id.UUIDGenerator",
        parameters = {
            @Parameter(
                name = "uuid_gen_strategy_class",
                value =
                "org.hibernate.id.uuid.CustomVersionOneStrategy"
            )
        }
    )
    @Column(name = "id", updatable = false, nullable = false)
    private UUID id;
```

```
    ...
}
```

Hibernate uses this mapping definition in the same way as in the previous example. It just uses a different algorithm to generate the UUID before it stores the new `Book` entity in the database.

```
Book b = new Book();
b.setTitle("Hibernate Tips");

em.persist(b);
```

```
05:57:35,777 DEBUG
[org.hibernate.event.internal.AbstractSaveEventListener] - Generated
identifier: c0a8b214-5abb-1aa1-815a-bbbaa51e0000, using strategy:
org.hibernate.id.UUIDGenerator
05:57:35,840 DEBUG [org.hibernate.SQL] -
    insert
    into
        Book
        (price, publishingDate, title, version, id)
    values
        (?, ?, ?, ?, ?)
05:57:35,845 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [1] as [DOUBLE] - [null]
05:57:35,846 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [2] as [DATE] - [null]
05:57:35,847 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [3] as [VARCHAR] - [Hibernate Tips]
05:57:35,848 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [4] as [INTEGER] - [0]
05:57:35,849 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [5] as [OTHER] - [c0a8b214-5abb-1aa1-815a-
bbbaa51e0000]
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `PrimaryKeyUUID` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

Hibernate also supports different options to generate numeric primary key values. You can read more about them in:

- How to use a custom database sequence to generate primary key values
- How to use an auto-incremented column to generate primary key values
- How to use a database table to generate primary key values

How to map a bidirectional many-to-one association

Problem

My table model contains a many-to-one association. How do I model it with Hibernate so that I can navigate it in both directions?

Solution

You need to model the association on both entities if you want to be able to navigate it in both directions. Consider this example. A book in an online bookstore can have multiple reviews. In your domain model, the `Book` entity has a one-to-many association to the `Review` entity, and the `Review` entity has a many-to-one relationship to the `Book` entity.

Let's begin with the `Review` entity, which is the owning side of the association in this example. That means that it defines the association and the `Book` entity just references it. The relationship consists of two mandatory and one optional part. The entity attribute of type `Book` and the `@ManyToOne` annotation are required. The attribute models the association, and the annotation declares the type of relationship. The `@JoinColumn` annotation is optional. It allows you to define the name of the foreign key column. I use it in the following code snippet to set the column name to `fk_book`.

If you don’t define the name yourself, Hibernate generates a name by combining the name of the association mapping attribute and the name of the primary key attribute of the associated entity. In this example, Hibernate would use `book_id` as the default column name.

```
@Entity
public class Review {

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

    @ManyToOne
    @JoinColumn(name = "fk_book")
    private Book book;

    ...
}
```

You also need to map the one-to-many association on the `Book` entity to make it bidirectional. As you can see in the following code snippet, this is done in a similar way as the many-to-one association.

```
@Entity
public class Book {

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

    @OneToMany(mappedBy = "book")
    private List<Review> reviews = new ArrayList<Review>();

    ...
}
```

You need an attribute that models the association, which is the `List<Review> reviews` attribute in this example and a `@OneToMany` annotation. Like in the table model, the bidirectional one-to-many association gets defined on the many side. The table on the many side stores the foreign key and its entity defines the association. It's similar for the entity mapping. You just need to reference the name of the association attribute of the many side as the value of the `mappedBy` attribute and Hibernate has all the information it needs.

That's all you need to do to define a bidirectional many-to-one association. You can now navigate it in both directions in your JPQL or Criteria API queries or on your domain objects.

```
b = em.find(Book.class, 1L);

List<Review> reviews = b.getReviews();
Assert.assertEquals(b, reviews.get(0).getBook());
```

Bidirectional associations are easy to use in queries, but they also require an additional step when you persist a new entity. You need to update the association on both sides when you add or remove an entity. You can see an example of it in the following code snippet, in which I first create a new `Review` entity and initialize its association to the `Book` entity. And after that, I also need to add the new `Review` entity to the `List` of reviews on the `Book` entity.

```
Book b = em.find(Book.class, 1L);

Review r = new Review();
r.setComment("This is a comment");
r.setBook(b);

b.getReviews().add(r);

em.persist(r);
```

Updating the associations on both entities is an error-prone task. Therefore, it's a good practice to provide a helper method that adds another entity to the many side of the association.

```
@Entity
public class Book {

    ...

    public void addReview(Review review) {
        this.reviews.add(review);
        review.setBook(this);
    }

    ...
}
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `AssociationBidirectionalManyToOne` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Bidirectional many-to-one associations are just one way to model relationships between entities. I show other options in:

- How to map an unidirectional many-to-one association
- How to map an unidirectional one-to-many association
- How to map a bidirectional one-to-one association

- How to map an unidirectional one-to-one association
- How to map a bidirectional many-to-many association
- How to map an unidirectional many-to-many association

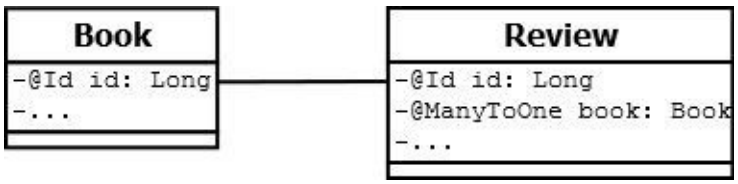
How to map an unidirectional many-to-one association

Problem

My table model contains a many-to-one association. I only want to model it on the many side. How do I do that with Hibernate?

Solution

Using an unidirectional many-to-one association is a typical approach for associations that contain a lot of entities on the many side of the relationship. It allows you to navigate it in the to-one direction but avoids performance issues that might occur if Hibernate has to load a huge number of entities to initialize the many side of the association.



You model the association only on the entities of the many side. Let's have a look at an example. A book in an online bookstore can have multiple reviews. In your domain model, you only model the many-to-one association on the `Review` entity. You can see an example of such a mapping in the following code snippet.

The association consists of two mandatory and one optional part. The entity attribute of type `Book` and the `@ManyToOne` annotation are required. The attribute models the association, and the annotation declares the kind of relationship. The `@JoinColumn` annotation is optional. It allows you to define

the name of the foreign key column. I use it in this example to set the column name to `fk_book`.

If you don't define the name yourself, Hibernate generates a name by combining the name of the association mapping attribute and the name of the primary key attribute of the associated entity. In this example, Hibernate would use `book_id` as the default column name.

```
@Entity
public class Review {

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

    @ManyToOne
    @JoinColumn(name = "fk_book")
    private Book book;

    ...
}
```

That's all you need to do to model an unidirectional many-to-one association. Like in the table model, the unidirectional many-to-to association gets defined on the many side. The table on the many side stores the foreign key and its entity defines the association. That makes it easy to model it as an unidirectional many-to-one association in your domain model.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `AssociationUnidirectionalManyToOne` module of the example

project. If you haven’t already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Unidirectional many-to-one associations are just one way to model relationships between entities. I show other options in:

- How to map a bidirectional many-to-one association
- How to map an unidirectional one-to-many association
- How to map a bidirectional one-to-one association
- How to map an unidirectional one-to-one association
- How to map a bidirectional many-to-many association
- How to map an unidirectional many-to-many association

How to map an unidirectional one-to-many association

Problem

My table model contains a one-to-many relationship. How do I model it as unidirectional association with Hibernate?

Solution

Modeling an unidirectional one-to-many association is not a very popular approach. It allows you to navigate it in the to-many direction but not in the to-one direction. Most developers prefer a bidirectional association or decide to model an unidirectional many-to-one association to avoid performance issues for relationships with huge numbers of entities.

You model the association only on the entities of the one side. Let’s have a look at an example. A book in an online bookstore can have multiple reviews. In your domain model, you only model the one-to-many association on the `Book` entity. You can see an example of such a mapping in the following code snippet. The `List<Review> reviews` attribute models the association and the `@OneToMany` annotation declares the kind of relationship.

```
@Entity
public class Book {

    @OneToMany
    private List<Review> reviews = new ArrayList<Review>();
```

```
    ...
}
```

That’s all you need to do to define a basic one-to-many association. You can now use the association in your queries and you can navigate it in your domain object tree. But there are two things you need to know before you model the association in this way. First of all, you need to keep in mind that you modeled the association only on the `Book` entity. When you persist a new `Review` entity, you need to update the associated `Book` entity as well.

```
Book b = em.find(Book.class, 1L);

Review r = new Review();
r.setComment("This is a comment");

b.getReviews().add(r);

em.persist(r);
```

More importantly, Hibernate uses a join table to persist the association.

```
05:50:37,181 DEBUG [org.hibernate.SQL] -
    insert
    into
        Review
        (comment, id)
    values
05:50:37,198 DEBUG [org.hibernate.SQL] -
    insert
    into
        Book_Review
        (Book_id, reviews_id)
    values
        (?, ?)
```

You can avoid that with a `@JoinColumn` annotation. It allows you to define the name of the foreign key column in the table mapped by the associated entity. In the following example, it tells Hibernate to use the `fk_book` column on the `review` table as the foreign key.

```
@Entity
public class Book {

    @OneToMany
    @JoinColumn(name = "fk_book")
    private List<Review> reviews = new ArrayList<Review>();

    ...
}
```

```
05:52:01,118 DEBUG [org.hibernate.SQL] -
    insert
    into
        Review
        (comment, id)
    values
        (?, ?)
05:52:01,125 DEBUG [org.hibernate.SQL] -
    update
        Review
    set
        fk_book=?
    where
        id=?
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `AssociationUnidirectionalOneToMany` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Unidirectional one-to-many associations are just one way to model relationships between entities. I show other options in:

- How to map a bidirectional many-to-one association
- How to map an unidirectional many-to-one association
- How to map a bidirectional one-to-one association
- How to map an unidirectional one-to-one association
- How to map a bidirectional many-to-many association
- How to map an unidirectional many-to-many association

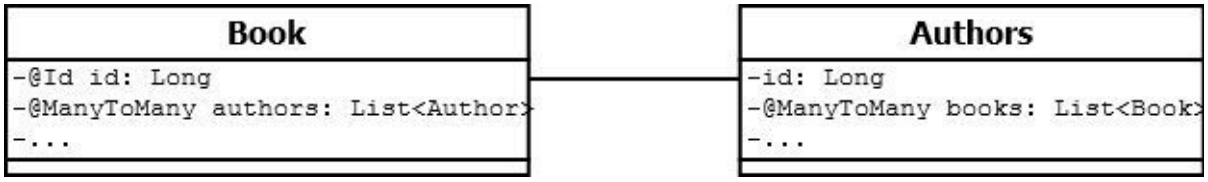
How to map a bidirectional many-to-many association

Problem

My table model contains a many-to-many association. How do I model it with Hibernate so that I can navigate it in both directions?

Solution

You need to model the association on both entities if you want to be able to navigate it in both directions. Let's have a look at an example. Multiple Authors can write multiple books and a book can be written by one or more authors. That's a typical many-to-many association, and you probably want to navigate it in both directions in your domain model and queries. You need to model it as a many-to-many association on the `Book` entity and the `Author` entity.



Let's begin with the `Book` entity, which is the owning side of the association in this example. That means that it defines the association and the `Author` entity just references it.

The relationship definition consists of two mandatory and one optional part. The entity attribute `List<Author> authors` and the `@ManyToMany` annotation are required. The attribute models the association, and the annotation declares the kind of relationship. The `@JoinTable` annotation is optional. It allows you to define the name of the join table and foreign key columns that store the many-to-many association. I use it in the following code snippet to set

the name of the join table to `book_author` and the names of the foreign key columns to `fk_book` and `fk_author`.

If you don't define the name yourself, Hibernate generates default table and column names. The default table name is the combination of both entity names. In this example, it would be `Book_Author`. The foreign key column name is generated by combining the name of the association mapping attribute and the name of the primary key attribute of the entity. These would be `books_id` and `authors_id` in this example.

```
@Entity
public class Book {

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

    @ManyToMany
    @JoinTable(
        name = "book_author",
        joinColumns = { @JoinColumn(name = "fk_book") },
        inverseJoinColumns = { @JoinColumn(name = "fk_author") })
    private List<Author> authors = new ArrayList<Author>();

    ...
}
```

You also need to map the many-to-many association on the `Author` entity to make it bidirectional. As you can see in the following code snippet, this is done in a similar way as on the `Book` entity. You need an attribute that models the association and a `@ManyToMany` annotation. In this example, it's the `List<Book> books` attribute that I annotated with a `@ManyToMany`

annotation. The association is already defined on the `Book` entity. You can therefore just reference the attribute on the `Book` entity in the `mappedBy` attribute and Hibernate uses the same definition.

```
@Entity
public class Author {

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

    @ManyToMany(mappedBy="authors")
    private List<Book> books = new ArrayList<Book>();

    ...
}
```

That's all you need to do to define a bidirectional many-to-many association. You can now navigate it in both directions in your JPQL or Criteria API queries or on your domain objects.

```
b = em.find(Book.class, 1L);

List<Author> authors = b.getAuthors();
```

Bidirectional associations are easy to use in queries, but they also require an additional step when you persist a new entity. You need to update the association on both sides when you add or remove an entity. You can see an example of it in the following code snippet in which I first create a new `Author` entity and add the `Book` entity to the `List` of `books`. And after that, I also need to add the new `Author` entity to the `List` of `authors` on the `Book` entity.

```
Book b = em.find(Book.class, 1L);

Author a = new Author();
a.setFirstName("Thorben");
a.setLastName("Janssen");

a.getBooks().add(b);
b.getAuthors().add(a);

em.persist(a);
```

Updating the associations on both entities is an error-prone task. It's therefore a good practice to provide helper methods for it.

```
@Entity
public class Author {

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

    @ManyToMany(mappedBy="authors")
    private List<Book> books = new ArrayList<Book>();

    public void addBook(Book b) {
        this.books.add(b);
        b.getAuthors().add(this);
    }

    public void removeBook(Book b) {
        this.books.remove(b);
        b.getAuthors().remove(this);
    }

    ...
}
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `AssociationBidirectionalManyToMany` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Bidirectional many-to-many associations are just one way to model relationships between entities. I show other options in:

- How to map an unidirectional many-to-many association
- How to map an unidirectional one-to-one association
- How to map a bidirectional one-to-one association
- How to map a bidirectional many-to-one association
- How to map an unidirectional many-to-one association
- How to map an unidirectional one-to-many association

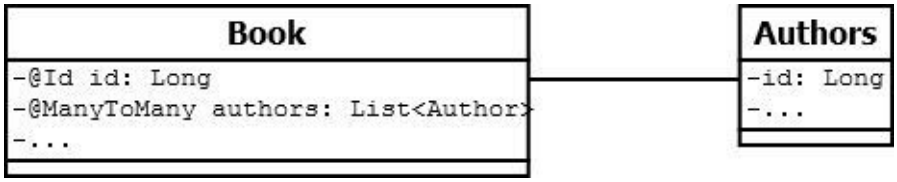
How to map an unidirectional many-to-many association

Problem

My table model contains a many-to-many association. I only need to navigate it in one direction. How do I model that with Hibernate?

Solution

You only need to model the association on the entity from which you want to navigate the relationship. Let's have a look at an example. Multiple Authors can write multiple books, and a book can be written by one or more authors. That's a typical many-to-many association. If you only want to navigate it from the `Book` to the `Author` entities, you only need to model it as a many-to-many association on the `Book` entity.



The relationship definition consists of two mandatory and one optional part. The entity attribute `List<Author> authors` and the `@ManyToMany` annotation are required. The attribute models the association, and the annotation declares the kind of relationship. The `@JoinTable` annotation is optional. It allows you to define the name of the join table and foreign key columns that store the many-to-many association. I use it in the following code snippet to set the name of the join table to `book_author` and the names of the foreign key columns to `fk_book` and `fk_author`.

If you don’t define the names yourself, Hibernate generates default table and column names. The default table name is the combination of both entity names. In this example, it would be `Book_Author`. The foreign key column is generated by combining the name of the association mapping attribute and the name of the primary key attribute of the entity. These would be `books_id` and `authors_id` in this example.

```
@Entity
public class Book {

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

    @ManyToMany
    @JoinTable(
        name = "book_author",
        joinColumns = { @JoinColumn(name = "fk_book") },
        inverseJoinColumns = { @JoinColumn(name = "fk_author") })
    private List<Author> authors = new ArrayList<Author>();

    ...
}
```

That’s all you need to do to define a unidirectional many-to-many association. You can now navigate it from the `Book` to the `Author` entities in your JPQL or Criteria API queries or on your domain objects.

```
b = em.find(Book.class, 1L);

List<Author> authors = b.getAuthors();
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `AssociationUnidirectionalManyToMany` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Unidirectional many-to-many associations are just one way to model relationships between entities. I show other options in:

- How to map a bidirectional many-to-many association
- How to map an unidirectional one-to-one association
- How to map a bidirectional one-to-one association
- How to map a bidirectional many-to-one association
- How to map an unidirectional many-to-one association
- How to map an unidirectional one-to-many association

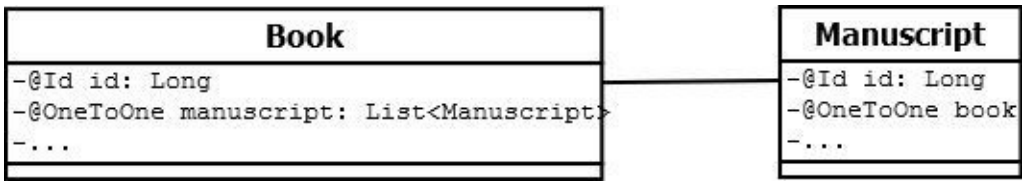
How to map a bidirectional one-to-one association

Problem

My table model contains a one-to-one association. How do I model it with Hibernate so that I can navigate it in both directions?

Solution

You need to model the association on both entities if you want to be able to navigate it in both directions. Let's have a look at an example. A book gets created from a manuscript. You could model this with a `Book` and a `Manuscript` entity and a one-to-one association between them. You need to model that with a one-to-one association on the `Book` entity and the `Manuscript` entity.



Let's begin with the `Manuscript` entity, which is the owning side of the association in this example. That means that it defines the relationship and the `Book` entity just references it.

The relationship definition consists of two mandatory and one optional part. The entity attribute `Book book` and the `@OneToOne` annotation are required. The attribute models the association, and the annotation declares the kind of relationship. The `@JoinColumn` annotation is optional. It allows you to define the name of the foreign key column that links the book to the manuscript. I use it

in the following code snippet to set the name of the foreign key column to `fk_book`. If you don't define the name yourself, Hibernate generates a name by combining the name of the association mapping attribute and the name of the primary key attribute of the associated entity. In this example, Hibernate would use `book_id` as the default column name.

```
@Entity
public class Manuscript {

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

    @OneToOne
    @JoinColumn(name = "fk_book")
    private Book book;

    ...
}
```

You also need to map the one-to-one association on the `Book` entity to make it bidirectional. As you can see in the following code snippet, this is done in a similar way as the one-to-one association on the `Manuscript` entity. You need an attribute that models the association and a `@OneToOne` annotation. In this example, it's the `Manuscript manuscript` attribute, which I annotated with a `@OneToOne` annotation. The association is already defined on the `Manuscript` entity. You can therefore just reference the attribute on the `Book` entity in the `mappedBy` attribute and Hibernate uses the same definition.

```
@Entity
public class Book {

    @Id
```

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

    @OneToOne(mappedBy = "book")
    private Manuscript manuscript;

    ...
}
```

That's all you need to do to define a bidirectional one-to-one association. You can now navigate it in both directions in your JPQL or Criteria API queries or on your domain objects.

```
Book b = em.find(Book.class, 1L);
Manuscript m = b.getManuscript();
Assert.assertEquals(b, m.getBook());
```

Bidirectional associations are easy to use in queries, but they also require an additional step when you persist a new entity. You need to update the association on both sides when you add or remove an entity. You can see an example of it in the following code snippet in which I first create a new `Manuscript` entity and initialize its association to the `Book` entity. And after that, I also need to set the new `Manuscript` entity on the `Book` entity.

```
Book b = em.find(Book.class, 1L);

Manuscript m = new Manuscript();
m.setBook(b);

b.setManuscript(m);

em.persist(m);
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `AssociationBidirectionalOneToOne` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Bidirectional one-to-one associations are just one way to model relationships between entities. I show other options in:

- How to map an unidirectional one-to-one association
- How to map a bidirectional many-to-one association
- How to map an unidirectional many-to-one association
- How to map an unidirectional one-to-many association
- How to map a bidirectional many-to-many association
- How to map an unidirectional many-to-many association

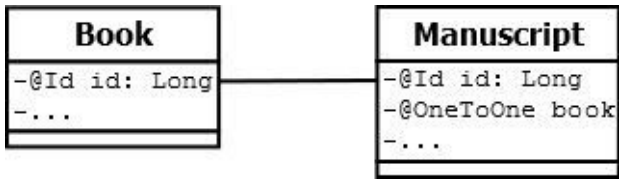
How to map an unidirectional one-to-one association

Problem

My table model contains an one-to-one association. I only need to navigate it one direction. How do I model that with Hibernate?

Solution

You only need to model the association on the entity from which you want to navigate the relationship. Let's have a look at an example. A book gets created from a manuscript. You could model this with a `Book` and a `Manuscript` entity and a one-to-one association between them. If you just want to navigate it from the `Manuscript` to the `Book` entity, you only need to model it as a one-to-one association on the `Manuscript` entity.



The relationship definition consists of two mandatory and one optional part. The entity attribute `Book book` and the `@OneToOne` annotation are required. The attribute models the association, and the annotation declares the kind of relationship. The `@JoinColumn` annotation is optional. It allows you to define the name of the foreign key column that links the book to the manuscript. I use it in the following code snippet to set the name of the foreign key column to `fk_book`. If you don't define the name yourself, Hibernate generates a name by combining the name of the association mapping attribute and the name of the

primary key attribute of the associated entity. In this example, Hibernate would use `book_id` as the default column name.

```
@Entity
public class Manuscript {

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

    @OneToOne
    @JoinColumn(name = "fk_book")
    private Book book;

    ...
}
```

That's all you need to do to define a unidirectional one-to-one association. You can now navigate from the `Manuscript` to the `Book` entity in your JPQL or Criteria API queries or on your domain objects.

```
Book b = em.find(Book.class, 1L);
Manuscript m = b.getManuscript();
Assert.assertEquals(b, m.getBook());
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `AssociationUnidirectionalOneToOne` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Unidirectional one-to-one associations are just one way to model relationships between entities. I show other options in:

- How to map a bidirectional one-to-one association
- How to map a bidirectional many-to-one association
- How to map an unidirectional many-to-one association
- How to map an unidirectional one-to-many association
- How to map a bidirectional many-to-many association
- How to map an unidirectional many-to-many association

ADVANCED MAPPINGS

The basic mappings described in the previous chapter provide good solutions for the most common use cases and are a good starting point, if you're new to Hibernate. But most real-world applications require more advanced mappings because their domain model uses custom data types or an inheritance structure and the database administrator defined views or database trigger to generate column values. All these use cases are not supported by the basic mappings. But JPA and Hibernate support a lot of advanced features that you can use to implement these use cases:

- How to map a view with Hibernate
- How to define a custom enum mapping
- How to map the Date and Time API with Hibernate 4.4
- How to map generated values
- How to calculate entity attributes with a @Formula
- How to cache preprocessed, non-persistent attributes
- How to automatically set an attribute before persisting it
- How to order the elements of a collection
- How to model a derived primary key with Hibernate

- How to model an association with additional attributes
- How to map an inheritance hierarchy to multiple tables
- How to map an inheritance hierarchy to one table

How to map a view with Hibernate

Problem

I have a read-only view that I want to use in associations and JPQL queries. How do I map it with Hibernate?

Solution

Database views, in general, are mapped in the same way as database tables. You just have to define an entity that maps the view with the specific name and one or more of its columns.

But the normal table mapping is not read-only, and you can use the entity to change its content.

Depending on the database you use and the definition of the view, you're not allowed to perform an update on the view content. You should therefore also prevent Hibernate from updating it.

You can easily achieve that by annotating your entity with Hibernate's `@Immutable` annotation.

```
@Entity
@Immutable
public class BookView {

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

    @Version
    @Column(name = "version")
```

```
    private int version;

    @Column
    private String title;
```

```
    @Column
    @Temporal(TemporalType.DATE)
    private Date publishingDate;

    @Column
    private String authors;

    ...

}
```

As a result, Hibernate performs an SQL `SELECT` statement to read the entity but it does not perform any `UPDATE` statements when you change an attribute.

```
BookView bv = em.find(BookView.class, 1L);
log.info(bv);
bv.setTitle("updated");
```

```
05:07:15,239 DEBUG [org.hibernate.SQL] - select bookview0_.id as
id1_3_0_, bookview0_.authors as authors2_3_0_,
bookview0_.publishingDate as publishi3_3_0_, bookview0_.title as
title4_3_0_, bookview0_.version as version5_3_0_ from BookView
bookview0_ where bookview0_.id=?
05:07:15,269 INFO  [org.thoughts.on.java.model.TestViewEntity] -
BookView [id=1, version=0, title=Hibernate Tips,
publishingDate=2017-04-04, authors=Thorben Janssen]
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `DatabaseViews` module of the example project. If you haven't already done

so, you can download it at http://www.hibernate-tips.com/download-examples.

How to define a custom enum mapping

Problem

I need to map enum values from a legacy database that don't match the enum standard mapping.

Or:

I don't want to face the drawbacks of JPA's standard enum mapping. How do I avoid them?

Solution

Since JPA 2.1, you can use an `AttributeConverter` to implement a custom mapping for your enums. It allows you to implement the conversion between a Java type and its database representation. You can use it to convert all basic attributes defined by entity classes, mapped superclasses, or embeddable classes. The only exceptions are id attributes, version attributes, relationship attributes and attributes annotated as `@Temporal` or `@Enumerated`.

In this example, I define a custom mapping for the `AuthorStatus` enumerations. I want to persist the value `PUBLISHED` as `P`, `SELF_PUBLISHED` as `S` and `NOT_PUBLISHED` as `N`.

```
public enum AuthorStatus {

    PUBLISHED, SELF_PUBLISHED, NOT_PUBLISHED;
}
```

The implementation of the custom mapping is simple. You just need to implement the `AttributeConverter` interface with its 2 methods `convertToDatabaseColumn` and `convertToEntityAttribute`. The implementation of these methods is pretty simple in this example. The `convertToDatabaseColumn` uses a `switch` statement to return the `String` representation of each value. The `convertToEntityAttribute` also uses a `switch` statement to implement the inverse mapping from the `String` representation to the `AuthorStatus` enum.

You also need to annotate the class with the `@Converter` annotation. The `autoApply` attribute of the `@Converter` annotation defines if Hibernate shall apply the `AttributeConverter` to all entity attributes of the given type. That is a good approach if you want to define a custom mapping for your enum values.

If you don't want to use the `AttributeConverter` for all entity attributes of type `AuthorStatus`, you can set the `autoApply` attribute to `false`. You then need to activate the converter for specific attributes. You can do that by annotating the entity attribute with a `@Convert(converter = AuthorStatusConverter.class)` annotation that references the class of the `AttributeConverter` you want to use.

```
@Converter(autoApply = true)
public class AuthorStatusConverter implements
AttributeConverter<AuthorStatus, String> {

    @Override
    public String convertToDatabaseColumn(AuthorStatus status) {
        switch (status) {
        case NOT_PUBLISHED:
            return "N";
```

```
        case PUBLISHED:
            return "P";

        case SELF_PUBLISHED:
            return "S";

        default:
            throw new IllegalArgumentException(
                "AuthorStatus ["+status+"] not supported.");
        }
    }
```

```
    @Override
    public AuthorStatus convertToEntityAttribute(String dbData) {
        switch (dbData) {
        case "N":
            return AuthorStatus.NOT_PUBLISHED;

        case "P":
            return AuthorStatus.PUBLISHED;

        case "S":
            return AuthorStatus.SELF_PUBLISHED;

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

The `autoApply` attribute makes this `AttributeConverter` easy to use. You just need to define an entity attribute of type `AuthorStatus` without a `@Enumerated` annotation, and Hibernate applies the conversion automatically.

```
@Entity
public class Author {
```

```
    private AuthorStatus status;

    ...
}
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `CustomEnumerationsMapping` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

JPA and Hibernate also support a standard mapping for enum values. You can read more about it in How to map an enum to a database column.

Custom enum mappings are just one use case for an `AttributeConverter`. You can also use it to map the classes of Java 8's Date and Time API with older Hibernate versions, as I show in How to map the Date and Time API with Hibernate 4.4.

How to map the Date and Time API with Hibernate 4.4

Problem

Hibernate 4.4 doesn't support the classes of Java 8's Date and Time API as attribute types. How do I persist these classes with Hibernate?

Solution

Since JPA 2.1, you can use an `AttributeConverter` to easily implement a custom mapping for unsupported data types. It allows you to implement the conversion between a Java type and its database representation. You can use it to convert all basic attributes defined by entity classes, mapped superclasses, or embeddable classes. The only exceptions are id attributes, version attributes, relationship attributes and attributes annotated as `@Temporal` or `@Enumerated`.

The following code snippet shows an `AttributeConverter<LocalDate, Date>` that converts an entity attribute of type `java.time.LocalDate` to a `java.sql.Date` to persist it in the database. When Hibernate reads the database column, it calls the `AttributeConverter` to transform the `java.sql.Date` into a `java.time.LocalDate`.

You just need to implement the `AttributeConverter` interface with its 2 methods `convertToDatabaseColumn` and `convertToEntityAttribute`. In this example, the implementation of these methods is pretty simple. The `java.sql.Date` class already provides the required methods to transform a `java.time.LocalDate` into a

`java.sql.Date` and to create a `java.time.LocalDate` from a `java.sql.Date`.

You also need to annotate the class with the `@Converter` annotation. Its `autoApply` attribute defines if Hibernate shall apply the AttributeConverter to all entity attributes of the given type. That is a good approach if you want to define custom mappings for the classes of the Date and Time API.

If you don't want to use the `AttributeConverter` for all entity attributes of type `LocalDate`, you can set the `autoApply` attribute to `false`. You then need to activate the converter for specific attributes. You can do that by annotating the entity attribute with a `@Convert(converter = LocalDateConverter.class)` annotation that references the class of the `AttributeConverter` you want to use.

```
@Converter(autoApply = true)
public class LocalDateConverter
        implements AttributeConverter<LocalDate, Date> {

    @Override
    public Date convertToDatabaseColumn(LocalDate attribute) {
        return Date.valueOf(attribute);
    }

    @Override
    public LocalDate convertToEntityAttribute(Date dbData) {
        return dbData.toLocalDate();
    }
}
```

The `autoApply` attribute makes this `AttributeConverter` easy to use. You just need to define an entity attribute of type `java.time.LocalDate` and Hibernate will applies the conversion automatically.

```
@Entity
public class Author {

    private LocalDate dateOfBirth;

    ...
}
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `AttributeConverterForDateAndTime` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Hibernate 5 supports the classes of the Date and Time API as `BasicType`. I show you how to use them in How to map classes of Java 8's Date and Time API.

You can also use `AttributeConverter` to implement custom enum mappings, as I show in How to define a custom enum mapping.

How to map generated values

Problem

My database administrator set up a trigger to generate the value of a database column. How do I map this column so that Hibernate retrieves the value after it gets generated?

Solution

You can annotate an entity attribute with `@Generated(GenerationTime value)`, to tell Hibernate that the database will generate the value of the attribute. The `GenerationTime` enum tells Hibernate when the database will generate the value. It can either do this `NEVER`, only on `INSERT` or `ALWAYS` (on insert and update). Hibernate then executes an additional query to retrieve the generated value from the database.

The following code snippet shows an example of such an entity mapping.

```
@Entity
public class Author {

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

    ...

}
```

As you can see in the log output, Hibernate now performs an additional query for each insert and update statement to retrieve the generated value.

```
// Transaction 1
em.getTransaction().begin();

Author a = new Author();
a.setFirstName("Thorben");
a.setLastName("Janssen");
em.persist(a);

em.getTransaction().commit();
log.info(a);

// Transaction 2
em.getTransaction().begin();

a = em.find(Author.class, a.getId());
a.setFirstName("Changed Firstname");

em.getTransaction().commit();
log.info(a);
```

```
12:06:36,349 DEBUG [org.hibernate.SQL] -
    insert
    into
        Author
        (firstName, lastName, version, id)
    values
        (?, ?, ?, ?)
12:06:36,353 DEBUG [org.hibernate.SQL] -
    select
        author_.lastUpdate as lastUpda4_0_
    from
        Author author_
    where
        author_.id=?
12:06:36,376 INFO  [org.thoughts.on.java.model.TestGeneratedColumn]
- Author [id=1, version=0, firstName=Thorben, lastName=Janssen,
lastUpdate=2017-03-09T12:06:36.322]
```

```
12:06:36,382 DEBUG [org.hibernate.SQL] -
    update
        Author
    set
        firstName=?,
        lastName=?,
        version=?
    where
        id=?
        and version=?
12:06:36,384 DEBUG [org.hibernate.SQL] -
    select
        author_.lastUpdate as lastUpda4_0_
    from
        Author author_
    where
        author_.id=?
12:06:36,387 INFO  [org.thoughts.on.java.model.TestGeneratedColumn]
- Author [id=1, version=1, firstName=Changed Firstname,
lastName=Janssen, lastUpdate=2017-03-09T12:06:36.383]
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `MapGeneratedColumns` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

How to calculate entity attributes with a @Formula

Problem

The value of one of the entity attributes needs to be calculated by a SQL expression. How do I map that with Hibernate?

Solution

You can use the `@Formula` annotation to provide a SQL snippet. Hibernate executes it when it fetches the entity from the database. The return value of the SQL expression gets mapped to a read-only entity attribute.

The following examples shows a `@Formula` that calculates the age of an author.

```
@Entity
public class Author {

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

    @Column
    private LocalDate dateOfBirth;

    @Formula(value = "date_part('year', age(dateOfBirth))")
    private int age;

    ...

    public int getAge() {
        return age;
    }
}
```

When Hibernate fetches an `Author` entity from the database, it adds the SQL snippet of the `@Formula` annotation to its SQL statement.

```
05:35:15,762 DEBUG [org.hibernate.SQL] – select author0_.id as
id1_0_, author0_.dateOfBirth as dateOfBi2_0_, author0_.firstName as
firstNam3_0_, author0_.lastName as lastName4_0_, author0_.version as
version5_0_, date_part('year', age(author0_.dateOfBirth)) as
formula0_ from Author author0_ where author0_.id=1
```

The @Formula annotation provides an easy way to map the result of an SQL snippet to an entity attribute. But it also has some downsides you should be aware of:

1. Hibernate executes the SQL snippet for every Author entity it fetches from the database. So better make sure, that you only use it for attributes you need in all of your use cases.
2. You need to provide a native SQL snippet to the `@Formula` annotation. This can affect the database portability of your application.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `Formula` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

If you don't want to execute the SQL snippet every time Hibernate loads the entity, you should make it a part of a custom query. This allows you to execute it only when you need it.

Here are a few tips that help you to define complex queries:

- How to create a JPQL query at runtime
- How to call a standard function in a JPQL query
- How to call a user-defined function in a JPQL query
- How to create a native SQL query at runtime

How to cache preprocessed, non-persistent attributes

Problem

I often use a value that gets calculated based on an entity attribute, like the age of a person. How do I cache this calculated attribute in the entity without storing it in the database?

Solution

There are different ways to provide a calculated value to the user of the entity:

1. You can use a `@Formula` to provide an SQL expression that returns the value.
2. You can use field access and calculate the value in a getter method.
3. You can use a `transient` entity attribute that stores the calculated value without persisting it in the database.

I already explained option 1 in How to calculate entity attributes with a @Formula. Here, I focus on options 2 and 3.

Calculate the value in a getter method

Option 2 is easy to implement. When you use field access, you can add additional getter methods to your entity. You can for example, add a `getAge()` method that calculates and returns the age of the `Author`.

This approach requires you to calculate the value for each call of the `getAge()` method. It is, therefore, not a good solution for complex calculations that you need to perform often.

```
@Entity
public class Author {

    ...

    @Column
    private LocalDate dateOfBirth;

    public int getCalculatedAge() {
        log.info("Calculate age");
        return Period.between(dateOfBirth,
                              LocalDate.now()).getYears();
    }
}
```

When you use this mapping, Hibernate doesn't select the value of the `age` attribute from the database and calculates it every time the `getAge` method gets called.

```
Author a = em.find(Author.class, 1L);
Assert.assertEquals(43, a.getCalculatedAge());
log.info(a.getFirstName() + " " + a.getLastName() + " is "
            + a.getCalculatedAge() + " years old.");
```

```
12:22:08,911 DEBUG [org.hibernate.SQL] - select author0_.id as
id1_0_0_, author0_.dateOfBirth as dateOfBi2_0_0_, author0_.firstName
as firstNam3_0_0_, author0_.lastName as lastName4_0_0_,
author0_.version as version5_0_0_ from Author author0_ where
author0_.id=?
12:22:08,916 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [1] as [BIGINT] - [1]
12:22:08,935 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([dateOfBi2_0_0_] : [DATE]) - [1973-04-12]
12:22:08,936 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([firstNam3_0_0_] : [VARCHAR]) - [John]
12:22:08,936 TRACE
```

```
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([lastName4_0_0_] : [VARCHAR]) - [Doe]
12:22:08,937 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([version5_0_0_] : [INTEGER]) - [0]
12:22:08,945 INFO  [org.thoughts.on.java.model.Author] - Calculate
age
12:22:08,957 INFO  [org.thoughts.on.java.model.Author] - Calculate
age
12:22:08,958 INFO  [org.thoughts.on.java.model.TestTransient] - John
Doe is 43 years old.
```

Use a transient entity attribute

The transient attribute approach requires a little more code but allows you to store the calculated value in an entity attribute without persisting it. The `@Transient` annotation tells Hibernate, and any other JPA implementation, to ignore the attribute when writing or reading the entity. You can, therefore, use it in your `getAge()` method to store the calculated result. This can be useful, if you need to perform a complex calculation that you don't want to repeat for each call.

```
@Entity
public class Author {

    ...

    @Column
    private LocalDate dateOfBirth;

    @Transient
    private Integer age;

    ...

    public int getAge() {
        if (this.age == null) {
            log.info("Calculate age");
```

```
            this.age = Period.between(dateOfBirth,
                                      LocalDate.now()).getYears();
        } else {
            log.info("Return cached value.");
        }

        return age;
    }
}
```

As you can see in the following example, Hibernate doesn't select the value of the age attribute from the database and it only gets calculated for the first call of the `getAge` method.

```
Author a = em.find(Author.class, 1L);
Assert.assertEquals(43, a.getAge());
log.info(a.getFirstName() + " " + a.getLastName() + " is "
            + a.getAge() + " years old.");
```

```
12:13:40,377 DEBUG [org.hibernate.SQL] - select author0_.id as
id1_0_0_, author0_.dateOfBirth as dateOfBi2_0_0_, author0_.firstName
as firstNam3_0_0_, author0_.lastName as lastName4_0_0_,
author0_.version as version5_0_0_ from Author author0_ where
author0_.id=?
12:13:40,382 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [1] as [BIGINT] - [1]
12:13:40,408 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([dateOfBi2_0_0_] : [DATE]) - [1973-04-12]
12:13:40,412 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([firstNam3_0_0_] : [VARCHAR]) - [John]
12:13:40,414 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([lastName4_0_0_] : [VARCHAR]) - [Doe]
12:13:40,416 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([version5_0_0_] : [INTEGER]) - [0]
12:13:40,425 INFO  [org.thoughts.on.java.model.Author] - Calculate
```

```
age
12:13:40,480 INFO  [org.thoughts.on.java.model.Author] - Return
cached value.
12:13:40,480 INFO  [org.thoughts.on.java.model.TestTransient] - John
Doe is 43 years old.
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `TransientAttributes` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

The `@Formula` annotation provides another option to calculate entity attribute values based on other database columns. You can read more about it in: How to calculate entity attributes with a @Formula.

How to automatically set an attribute before persisting it

Problem

I want to initialize an entity attribute automatically before it gets persisted. How do I execute custom code before Hibernate persists an entity?

Solution

The JPA specification defines a set of callback annotations to trigger method calls for certain lifecycle events. If you want to initialize an entity attribute before it gets persisted, you just have to do 2 things:

1. Add a method to the entity that initializes the attribute.
2. Annotate this method with `@PrePersist` so that Hibernate calls it before it persists the entity.

You can see an example of such a method in the following code snippet.

```
@Entity
public class Author {

        ...

        @PrePersist
        private void initializeCreatedAt() {
                this.createdAt = LocalDateTime.now();
                log.info("Set createdAt to "+this.createdAt);
        }
}
```

Hibernate calls this method before it persists the new Author entity and trigger the initialization of the `createdAt` attribute. You can see that in the log

output, when you persist a new Author entity.

```
Author a = new Author();
a.setFirstName("Thorben");
a.setLastName("Janssen");
em.persist(a);
```

Hibernate calls the `initializeCreatedAt` method before it selects the primary key value from the database. It then persists the `createdAt` attribute value in the database.

```
05:34:00,205 INFO  [org.thoughts.on.java.model.Author] - Set
createdAt to 2017-03-03T05:34:00.198
05:34:00,211 DEBUG [org.hibernate.SQL] -
    select
        nextval ('hibernate_sequence')
05:34:00,260 DEBUG [org.hibernate.SQL] -
    insert
    into
        Author
        (createdAt, firstName, lastName, version, id)
    values
        (?, ?, ?, ?, ?)
05:34:00,267 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [1] as [TIMESTAMP] - [2017-03-03T05:34:00.198]
05:34:00,269 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [2] as [VARCHAR] - [Thorben]
05:34:00,270 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [3] as [VARCHAR] - [Janssen]
05:34:00,271 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [4] as [INTEGER] - [0]
05:34:00,272 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [5] as [BIGINT] - [1]
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `PrePersistLifecycleEvent` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

The JPA specification also defines callback annotations for other entity lifecycle events. You can, for example, use the `@PreRemove` annotation to execute custom code before Hibernate deletes an entity, as I do in this post: How to implement a soft delete with Hibernate.

How to order the elements of a collection

Problem

How do I order the elements of a collection without writing my own query?

Solution

JPA supports the `@OrderBy` annotation. You can use it to define the order in which associated entities shall be retrieved from the database.

> Hibernate applies the ordering every time it loads the associated entities from the database. That slows down the query, and you should make sure that you always need to retrieve the entities in the defined order. Otherwise, you should use a use case-specific query that returns the entities in the defined order.

You can apply the `@OrderBy` annotation to a relationship attribute and define the ordering in the same way as in a JPQL query. You have to provide one or more entity attributes as a comma-separated list. Hibernate applies an ascending order by default. But you can also define the preferred order for each of the attributes. You just need to add `ASC` to define an ascending order or `DESC` for an descending order behind the attribute name.

I use this annotation in the following code snippet to retrieve the `Authors` of a `Book` in the ascending order of their `lastName` attribute.

```
@ManyToMany
@JoinTable(name="BookAuthor",
           joinColumns={@JoinColumn(name="bookId",
```

```
                                        referencedColumnName="id")},
            inverseJoinColumns={@JoinColumn(name="authorId",
                                        referencedColumnName="id")})
@OrderBy(value = "lastName ASC")
private Set<Author> authors = new HashSet<Author>();
```

Hibernate uses the value of the annotation to create an `ORDER BY` statement when it fetches the related entities from the database.

```
05:22:13,930 DEBUG [org.hibernate.SQL] –
select authors0_.bookId as bookId1_2_0_,
    authors0_.authorId as authorId2_2_0_,
    author1_.id as id1_0_1_,
    author1_.firstName as firstNam2_0_1_,
    author1_.lastName as lastName3_0_1_,
    author1_.version as version4_0_1_
from BookAuthor authors0_ inner join Author author1_
    on authors0_.authorId=author1_.id
where authors0_.bookId=?
order by author1_.lastName asc
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `OrderRelationships` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

How to model a derived primary key with Hibernate

Problem

My database table uses a derived primary key that includes the primary key of an associated entity. How do I model such a primary key with Hibernate?

Solution

You can model a derived primary key with an embedded id and a `@MapsId` annotation on the association whose primary key is part of the derived identifier. The following code snippet shows an example of such a mapping.

```
@Entity
public class Review {

    @EmbeddedId
    private ReviewId id;

    private String comment;

    @ManyToOne
    @JoinColumn(name = "fk_book")
    @MapsId("bookId")
    private Book book;

    ...
}
```

The `id` attribute models the primary key of the `Review` entity. It is of type `ReviewId`, which is an embeddable and shown in the following code snippet.

The second important part of this mapping is the `@MapsId` annotation on the `Book book` association. You can use this annotation with many-to-one and

one-to-one associations. It tells Hibernate to use the primary key of the to-one side of the association as a primary key attribute of the to-many side of the association. It also provides the name of the embedded id attribute to which it shall be mapped. In this example, the `id` attribute of the `Book` entity becomes part of the primary key of the `Review` entity and gets mapped to the `bookId` attribute of the `ReviewId` class.

You can see the `ReviewId` class in the following code snippet. It models the primary key with the attributes `bookId` and `userName`. It is used as an embedded id and needs to fulfill the requirements of a primary key class. A primary key class must be public, have a public default constructor, be serializable and implement the equals and hashCode methods.

```
@Embeddable
public class ReviewId implements Serializable {

    private static final long serialVersionUID =
        -5073745645379676235L;

    private String userName;

    private Long bookId;

    public ReviewId() {

    }

    public ReviewId(String userName, Long bookId) {
        this.userName = userName;
        this.bookId = bookId;
    }

    public String getUserName() {
        return userName;
    }
```

```
    public void setUserName(String userName) {
        this.userName = userName;
    }

    public Long getBookId() {
        return bookId;
    }

    public void setBookId(Long bookId) {
        this.bookId = bookId;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result
                + ((bookId == null) ? 0 : bookId.hashCode());
        result = prime * result
                + ((userName == null) ? 0 : userName.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;

        if (obj == null)
            return false;

        if (getClass() != obj.getClass())
            return false;

        ReviewId other = (ReviewId) obj;

        return Objects.equals(bookId, other.bookId)
                && Objects.equals(userName, other.userName);
    }
}
```

The following code snippet shows an example that persists a new `Review` entity. Due to the `@MapsId` annotation, Hibernate automatically initializes the primary key attribute `bookId` when you call the `setBook(Book b)` method. That requires you to initialize the `id` attribute before you set the `Book` association.

```
Review r = new Review();
r.setId(new ReviewId());
r.getId().setUserName("peter");
r.setBook(em.find(Book.class, 1L));

r.setComment("This is a comment");

em.persist(r);
```

The `ReviewId` class only contains the `bookId` attribute of type `Long` and not the association to the `Book` entity. This makes it easy to instantiate a new `ReviewId` object, if you want to lookup a `Review` entity.

```
Review r = em.find(Review.class, new ReviewId("peter", 1L));
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `DerivedPrimaryKey` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Derived primary keys are often used by entities that model the join tables of many-to-many associations. I use one in the Hibernate tip How to model an

association with additional attributes.

How to model an association with additional attributes

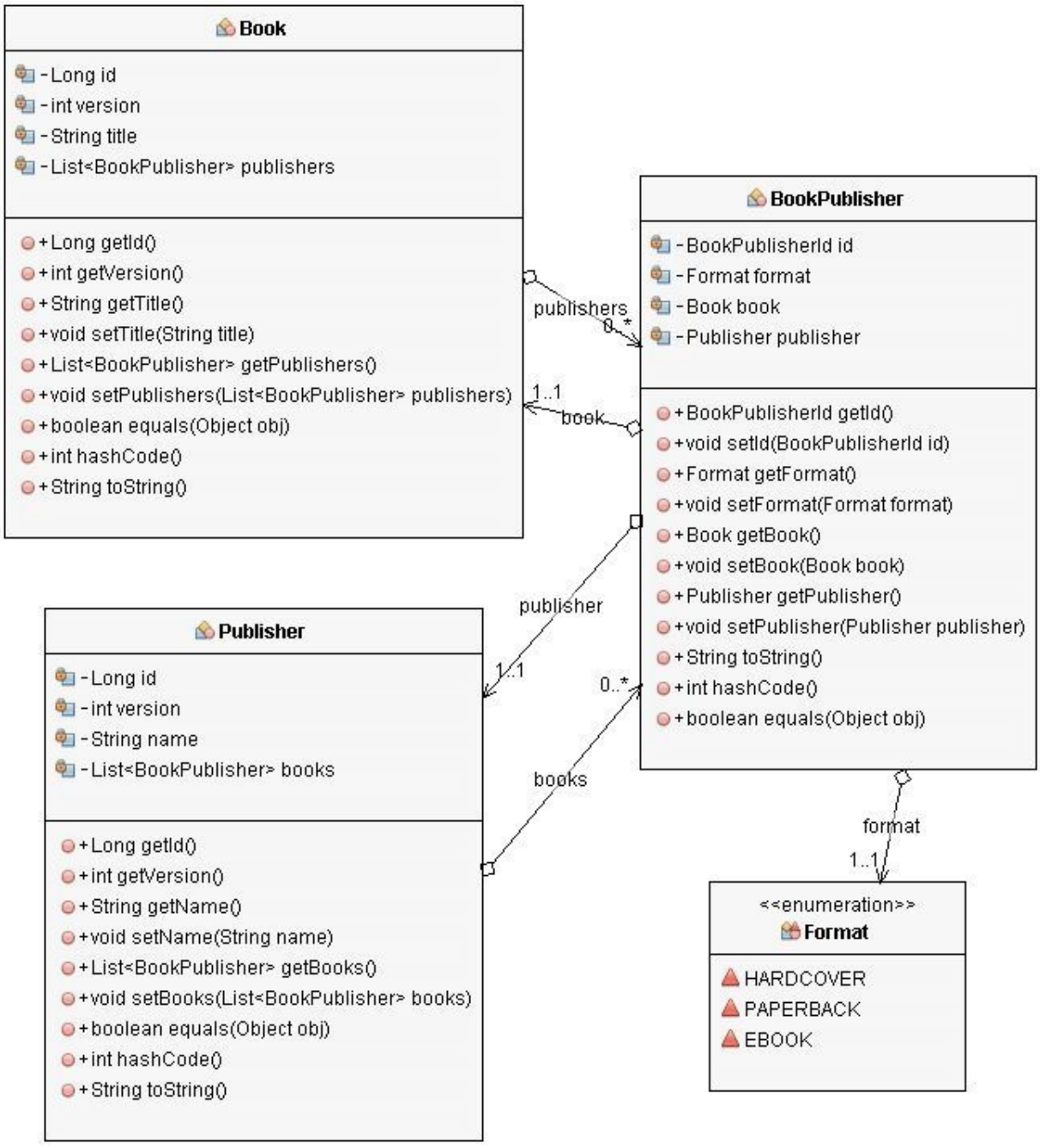
Problem

I need to persist additional attributes for my many-to-many association. How do I do that with Hibernate?

Solution

Modelling a many-to-many association with JPA and Hibernate is simple. You just need an entity attribute and a `@ManyToMany` annotation. Hibernate internally maps the relationship to the join table you need in your table model. But this approach has one downside. It doesn't allow you to map any columns of the join table. If you want to do that, you need to model it in the same way as you do it in the database. You need to split the many-to-many association into an entity that maps the join table and has two many-to-one relationships to the associated entities.

The following diagram shows the domain model of a bookstore that uses an association with additional attributes. There are books in multiple formats (e.g. hardcover, paperback, ebook) and each format was published by a different publisher. The `Book` and `Publisher` entities model the two main domain objects. The `BookPublisher` entity models the association between them and persists the `Format` as an additional attribute.



There is nothing special about the `Book` and `Publisher` entity in this example. They model the properties of the domain model and have a one-to-many association to the `BookPublisher` entity.

```
@Entity
public class Book {

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

    @Version
    private int version;

    private String title;

    @OneToMany(mappedBy = "book")
    private List<BookPublisher> publishers =
        new ArrayList<BookPublisher>();

    ...
}
```

```
@Entity
public class Publisher {

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

    @Version
    private int version;

    private String name;

    @OneToMany(mappedBy = "publisher")
    private List<BookPublisher> books =
        new ArrayList<BookPublisher>();

    ...
}
```

The `BookPublisher` entity models the association between the `Book` and the `Publisher` entity and stores the additional `Format format` attribute. It's a common practice to use a composite primary key with the derived primary keys of the associated entities. I explain derived primary keys and how to model them with an embedded id in How to model a derived primary key with Hibernate.

```
@Entity
public class BookPublisher {

    @EmbeddedId
    private BookPublisherId id;

    @Enumerated(EnumType.STRING)
    private Format format;

    @ManyToOne
    @JoinColumn(name = "fk_book")
    @MapsId("bookId")
    private Book book;

    @ManyToOne
    @JoinColumn(name = "fk_publisher")
    @MapsId("publisherId")
    private Publisher publisher;

    ...
}
```

```
@Embeddable
public class BookPublisherId implements Serializable {

    private static final long serialVersionUID =
        -3287715633608041039L;

    private Long bookId;

    private Long publisherId;
```

```
    public BookPublisherId() {

    }

    public BookPublisherId(Long bookId, Long publisherId) {
        this.bookId = bookId;
        this.publisherId = publisherId;
    }

    public Long getBookId() {
        return bookId;
    }

    public Long getPublisherId() {
        return publisherId;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result
            + ((bookId == null) ? 0 : bookId.hashCode());
        result = prime * result
            + ((publisherId == null) ? 0 : publisherId.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        BookPublisherId other = (BookPublisherId) obj;

        return Objects.equals(bookId, other.bookId)
                && Objects.equals(publisherId,
                                  other.getPublisherId());
    }
```

```
    @Override
    public String toString() {
        return "BookPublisherId [bookId=" + bookId
                + ", publisherId=" + publisherId + "]";
    }
}
```

When you want to persist the association between a `Book` and a `Publisher` entity, you need to create a new `BookPublisher` entity and associate it with a `Book` and a `Publisher` entity.

```
Book b = em.find(Book.class, 1L);
Publisher p = em.find(Publisher.class, 1L);

BookPublisher bp = new BookPublisher();
bp.setId(new BookPublisherId());
bp.setBook(b);
bp.setPublisher(p);
bp.setFormat(Format.PAPERBACK);

em.persist(bp);
```

```
16:37:31,898 DEBUG [org.hibernate.SQL] -
    select
        book0_.id as id1_0_0_,
        book0_.title as title2_0_0_,
        book0_.version as version3_0_0_
    from
        Book book0_
    where
        book0_.id=?
16:37:31,902 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [1] as [BIGINT] - [1]
16:37:31,916 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([title2_0_0_] : [VARCHAR]) - [Hibernate Tips]
16:37:31,917 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
```

```
([version3_0_0_] : [INTEGER]) - [0]
16:37:31,937 DEBUG [org.hibernate.SQL] -
    select
        publisher0_.id as id1_2_0_,
        publisher0_.name as name2_2_0_,
        publisher0_.version as version3_2_0_
    from
        Publisher publisher0_
    where
        publisher0_.id=?
16:37:31,938 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [1] as [BIGINT] - [1]
16:37:31,939 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([name2_2_0_] : [VARCHAR]) - [Thoughts on Java]
16:37:31,940 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([version3_2_0_] : [INTEGER]) - [0]
16:37:31,979 DEBUG [org.hibernate.SQL] -
    insert
    into
        BookPublisher
        (format, fk_book, fk_publisher)
    values
        (?, ?, ?)
16:37:31,979 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [2] as [BIGINT] - [1]
16:37:31,979 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [3] as [BIGINT] - [1]
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `AssociationsWithAttributes` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

It is a common practice to model the primary key of the association table and its entity as a derived primary key. I explain the required mapping in more detail in How to model a derived primary key with Hibernate.

How to map an inheritance hierarchy to multiple tables

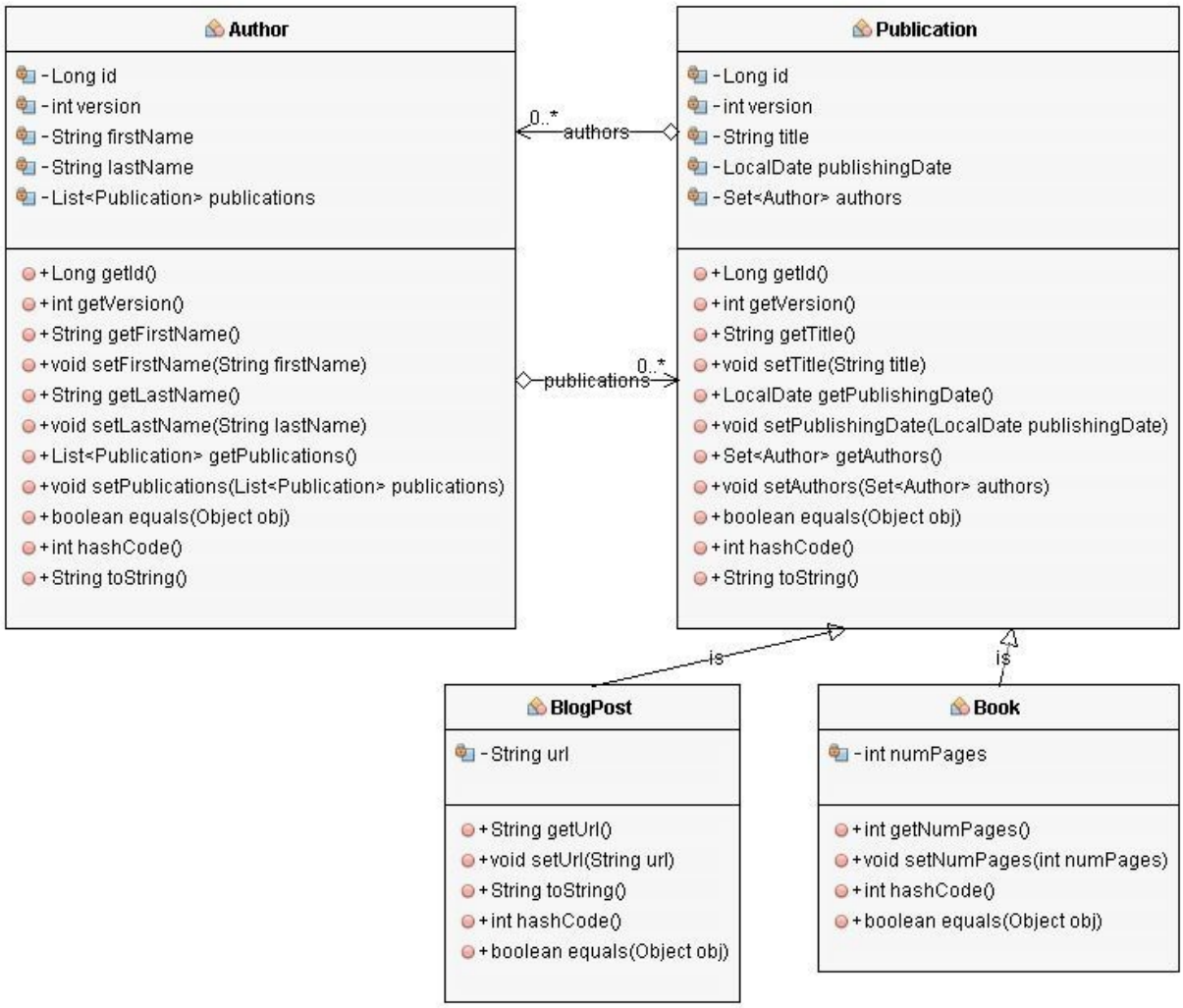
Problem

My database contains multiple tables that I want to map to an inheritance hierarchy of entities. How do I define such a mapping?

Solution

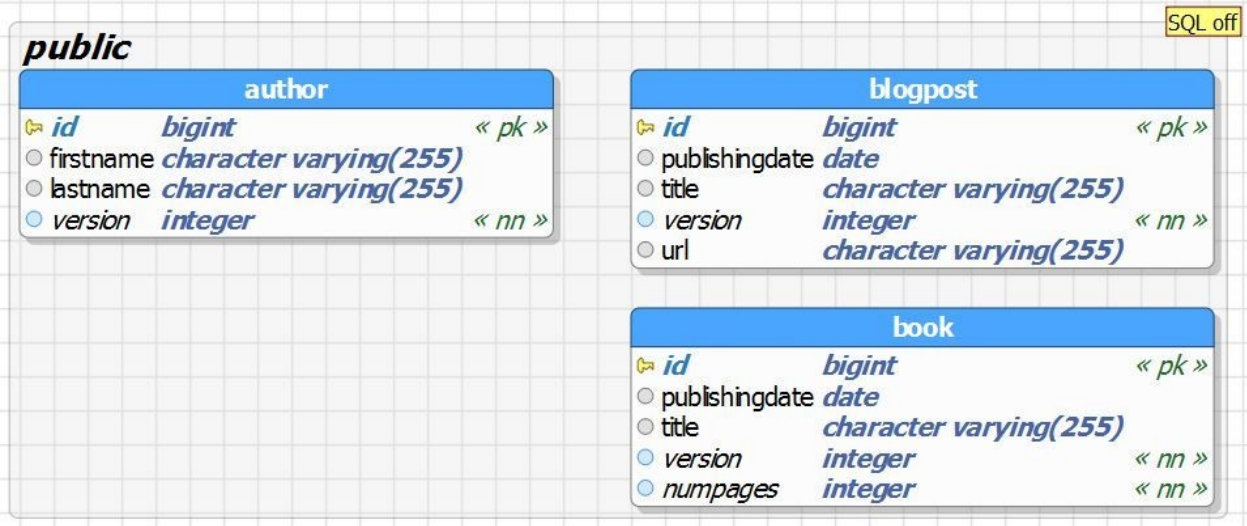
JPA and Hibernate support different inheritance strategies that allow you to map the entities to different table structures. The `MappedSuperclass` approach and the strategies `TablePerClass` and `Joined` map an inheritance hierarchy of entities to multiple tables. Each of them maps the entities to a different table structure with its advantages and disadvantages. You need to decide for your particular use case which approach you want to use.

Let's take a look at the entity model that I use in all examples of this Hibernate tip before I show you the different inheritance strategies. `Authors` can write different kinds of `Publications`, like `Books` and `BlogPosts`. The `Publication` class is the super class of the `Book` and `BlogPost` classes.



MappedSuperclass

The `MappedSuperclass` approach maps each entity class to a database table but not the superclass. In this example, it maps the entity classes `Author`, `BlogPost` and `Book` to the database tables `author`, `blogpost` and `book`. The `Publication` class does not get mapped to any database table.



You define this mapping by annotating the superclass with the `@MappedSuperclass` annotation but not with a `@Entity` annotation. The superclass is not an entity. It just defines the attributes that are shared by the subclasses.

```
@MappedSuperclass
public abstract class Publication {

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

    @Version
    private int version;

    private String title;

    private LocalDate publishingDate;

    ...
}
```

The subclasses only need to extend the superclass and are annotated with a `@Entity` annotation. The mapping of the inheritance hierarchy doesn't require

any additional annotations.

```
@Entity
public class Book extends Publication {

    private int numPages;

    ...
}
```

The `MappedSuperclass` approach enables Hibernate to select `Book` or `BlogPost` entities with a simple SQL query. But it doesn't support any polymorphic queries.

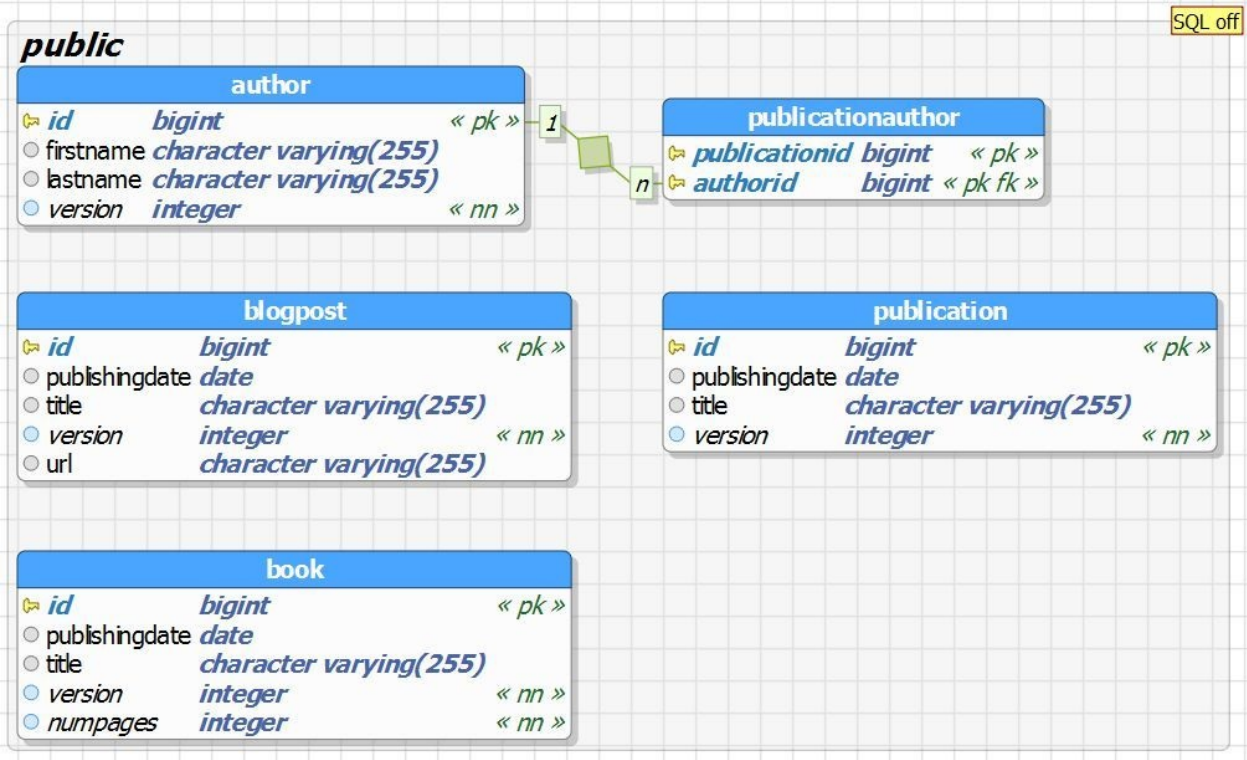
```
TypedQuery<Book> q = em.createQuery(
    "SELECT b FROM Book b WHERE b.id = :id", Book.class);
q.setParameter("id", 1L);
b = q.getSingleResult();
```

```
13:49:00,420 DEBUG [org.hibernate.SQL] -
    select
        book0_.id as id1_2_,
        book0_.publishingDate as publishi2_2_,
        book0_.title as title3_2_,
        book0_.version as version4_2_,
        book0_.numPages as numPages5_2_
    from
        Book book0_
    where
        book0_.id=?
```

TablePerClass

The `TablePerClass` strategy maps each concrete entity class of the hierarchy, including the superclass, to its own database table. In contrast to the

`MappedSuperclass` approach, the superclass is now also an entity which gets mapped to a database table and can be used in queries and associations.



That means in the current example, that Hibernate maps the `Author`, `Publication`, `Book` and `BlogPost` entities to database tables with the same name and a column for each entity attribute.

If you want to use this inheritance strategy, you need to annotate the superclass with an `@Inheritance` annotation and provide the `InheritanceType.TABLE_PER_CLASS` as the value of the `strategy` attribute.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Publication {

    @Id
```

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

    @Version
    private int version;

    private String title;

    private LocalDate publishingDate;

    @ManyToMany
    @JoinTable(
        name="PublicationAuthor",
        joinColumns={@JoinColumn(name="publicationId",
                                 referencedColumnName="id")},
        inverseJoinColumns={@JoinColumn(name="authorId",
                                 referencedColumnName="id")})
    private Set<Author> authors = new HashSet<Author>();

    ...
}
```

The subclasses only need to extend the superclass and are annotated with a `@Entity` annotation.

```
@Entity
public class Book extends Publication {

    private int numPages;

    ...
}
```

The `TablePerClass` strategy allows efficient queries as long as you select only one kind of entity. But as you can see in the following log output, polymorphic queries and associations require complex join statements and should be avoided.

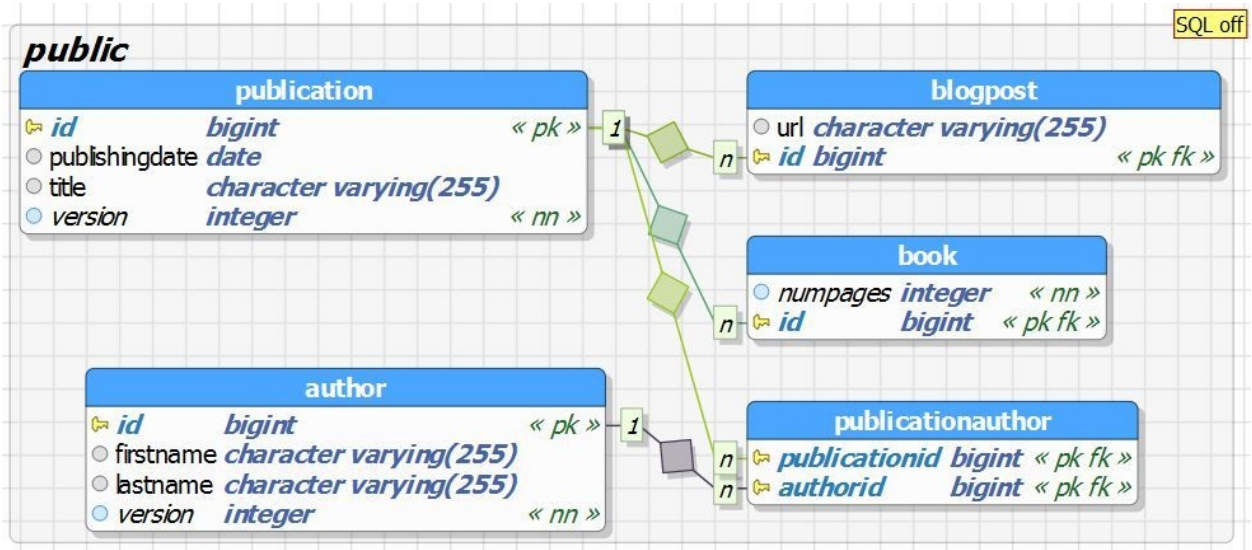
```
Author a = em.find(Author.class, 1L);
List<Publication> publications = a.getPublications();
```

```
05:41:00,338 DEBUG [org.hibernate.SQL] -
    select
        publicatio0_.authorId as authorId2_4_0_,
        publicatio0_.publicationId as publicat1_4_0_,
        publicatio1_.id as id1_3_1_,
        publicatio1_.publishingDate as publishi2_3_1_,
        publicatio1_.title as title3_3_1_,
        publicatio1_.version as version4_3_1_,
        publicatio1_.numPages as numPages1_2_1_,
        publicatio1_.url as url1_1_1_,
        publicatio1_.clazz_ as clazz_1_
    from
        PublicationAuthor publicatio0_
    inner join
        (
            select
                id,
                publishingDate,
                title,
                version,
                null::int4 as numPages,
                null::varchar as url,
                0 as clazz_
            from
                Publication
            union
            all select
                id,
                publishingDate,
                title,
                version,
                numPages,
                null::varchar as url,
                1 as clazz_
            from
                Book
            union
            all select
```

```
                id,
                publishingDate,
                title,
                version,
                null::int4 as numPages,
                url,
                2 as clazz_
            from
                BlogPost
        ) publicatio1_
            on publicatio0_.publicationId=publicatio1_.id
    where
        publicatio0_.authorId=?
```

Joined

The `Joined` strategy maps each entity of the hierarchy, including the superclass, to a database table. But in contrast to the `TablePerClass` approach, the table of the superclass contains all columns shared by the subclasses. That makes the `book` and `blogpost` table of the current example a lot smaller. They only consist of a primary key column and a column to persist the additional attribute of each entity.



You specify the `Joined` table strategy in a similar way as the `TablePerClass` strategy. You need to add an `@Inheritance` annotation to the superclass and set the `strategy` attribute to `InheritanceType.JOINED`.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Publication {

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

    @Version
    private int version;

    private String title;

    private LocalDate publishingDate;

    @ManyToMany
    @JoinTable(name="PublicationAuthor",
        joinColumns={@JoinColumn(name="publicationId",
                                 referencedColumnName="id")},
        inverseJoinColumns={@JoinColumn(name="authorId",
                                 referencedColumnName="id")})
    private Set<Author> authors = new HashSet<Author>();

    ...
}
```

Like for the previously discussed mapping strategies, the subclasses only need to extend the superclass and are annotated with a `@Entity` annotation.

```
@Entity
public class Book extends Publication {

    private int numPages;
```

```
    ...
}
```

The `Joined` table strategy maps the entity attributes defined by the superclass to columns in the `publication` table and the attributes defined by the `Book` or `BlogPost` entity to columns in the `book` or `blogpost` table. That makes the tables of the subclasses smaller and polymorphic queries easier. But all select statements require at least one `JOIN` clause that joins the table of the superclass with the table of the selected subclass.

```
TypedQuery<Book> q = em.createQuery(
    "SELECT b FROM Book b WHERE b.id = :id", Book.class);
q.setParameter("id", 1L);
b = q.getSingleResult();
```

```
14:09:06,926 DEBUG [org.hibernate.SQL] -
    select
        book0_.id as id1_3_,
        book0_1_.publishingDate as publishi2_3_,
        book0_1_.title as title3_3_,
        book0_1_.version as version4_3_,
        book0_.numPages as numPages1_2_
    from
        Book book0_
    inner join
        Publication book0_1_
            on book0_.id=book0_1_.id
    where
        book0_.id=?
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `InheritanceMappedSuperclass`, `InheritanceTablePerClass`

and `InheritanceJoined` modules of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

You can also map all entities of the inheritance hierarchy to the same database table. I show you how to do that in How to map an inheritance hierarchy to one table.

How to map an inheritance hierarchy to one table

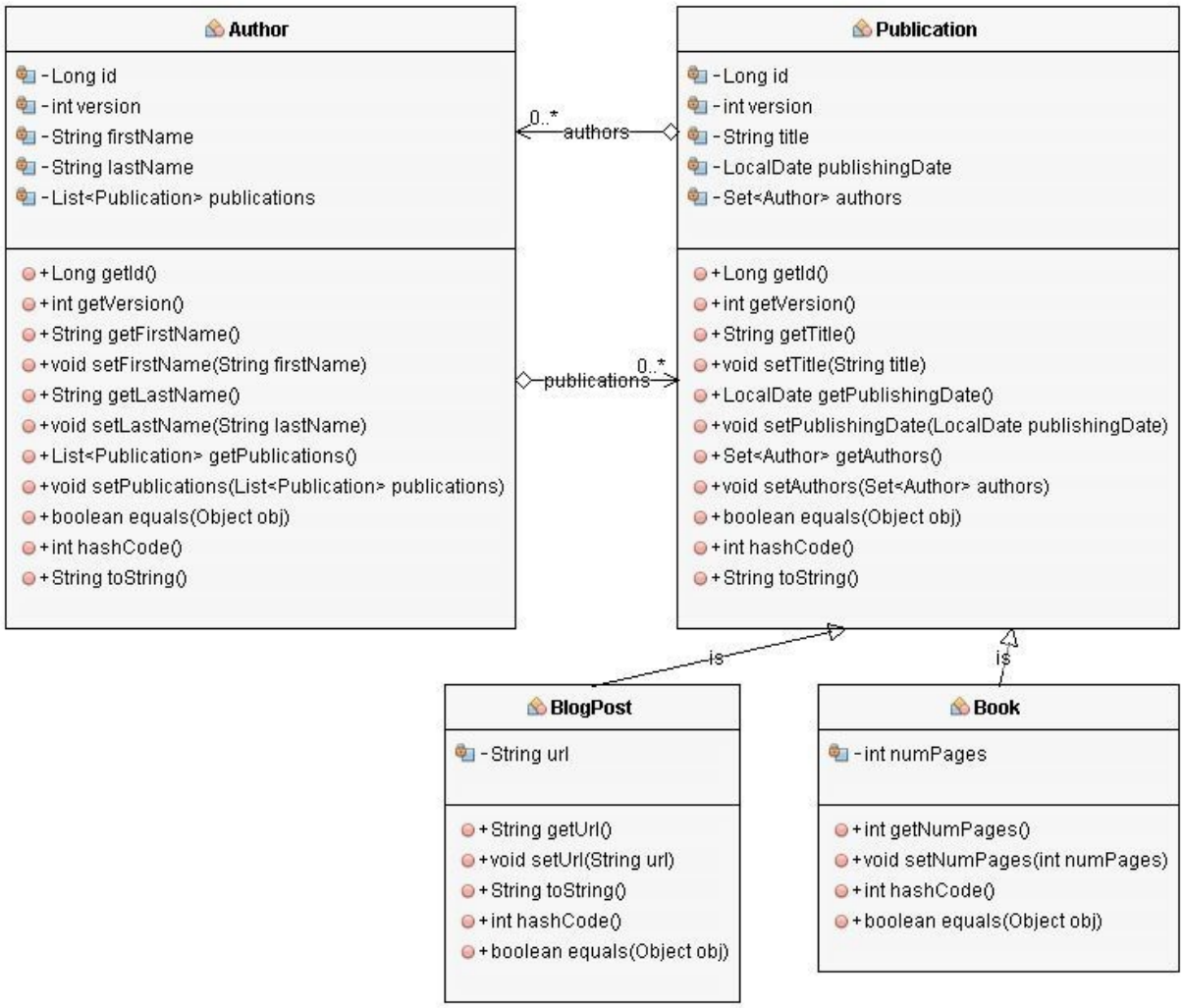
Problem

My database contains one table, which I want to map to an inheritance hierarchy of entities. How do I define such a mapping?

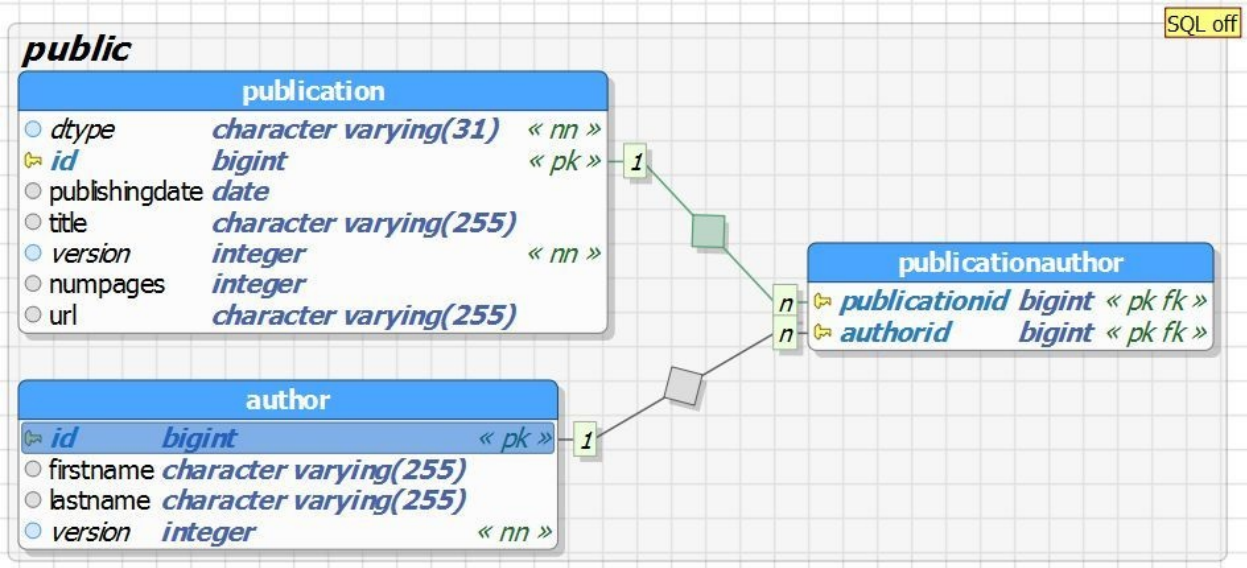
Solution

JPA and Hibernate support different inheritance strategies which allow you to map the entities to different table structures. The `SingleTable` strategy is one of them and maps an inheritance hierarchy of entities to a single database table.

Let's have a look at the entity model before I explain the details of the `SingleTable` strategy. `Authors` can write different kinds of `Publications`, like `Books` and `BlogPosts`. The `Publication` class is the super class of the `Book` and `BlogPost` classes.



The `SingleTable` strategy maps the three entities of the inheritance hierarchy to the `publication` table.



If you want to use this inheritance strategy, you need to annotate the superclass with an `@Inheritance` annotation and provide the `InheritanceType.SINGLE_TABLE` as the value of the `strategy` attribute.

You can also annotate the superclass with a `@DiscriminatorColumn` annotation to define the name of the discriminator value. Hibernate uses this value to determine the entity to which it has to map a database record. If you don't define a discriminator column, as I do in the following code snippet, Hibernate, and all other JPA implementations use the column `DTYPE`.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Publication {

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

    @Version
    private int version;
```

```
    private String title;

    private LocalDate publishingDate;

    @ManyToMany
    @JoinTable(name="PublicationAuthor",
        joinColumns={@JoinColumn(name="publicationId",
                                 referencedColumnName="id")},
        inverseJoinColumns={@JoinColumn(name="authorId",
                                 referencedColumnName="id")})
    private Set<Author> authors = new HashSet<Author>();

    ...
}
```

The subclasses need to extend the superclass, and you need to annotate them with a `@Entity` annotation. The JPA specification also recommends to annotate it with a `@DiscriminatorValue` annotation to define the discriminator value for this entity class. If you don't provide this annotation, your JPA implementation generates a discriminator value. But the JPA specification doesn't define how to generate the discriminator value, and your application might not be portable to other JPA implementations. Hibernate uses the simple entity name as the discriminator.

```
@Entity
@DiscriminatorValue("Book")
public class Book extends Publication {

    private int numPages;

    ...
}
```

The `SingleTable` strategy doesn't require Hibernate to generate any complex queries if you want to select a specific entity, perform a polymorphic query or

traverse a polymorphic association. All entities are stored in the same table, and Hibernate can select them from there without an additional `JOIN` clause.

```
Author a = em.find(Author.class, 1L);
List<Publication> publications = a.getPublications();
```

```
14:29:53,723 DEBUG [org.hibernate.SQL] -
    select
        publicatio0_.authorId as authorId2_2_0_,
        publicatio0_.publicationId as publicat1_2_0_,
        publicatio1_.id as id2_1_1_,
        publicatio1_.publishingDate as publishi3_1_1_,
        publicatio1_.title as title4_1_1_,
        publicatio1_.version as version5_1_1_,
        publicatio1_.numPages as numPages6_1_1_,
        publicatio1_.url as url7_1_1_,
        publicatio1_.DTYPE as DTYPE1_1_1_
    from
        PublicationAuthor publicatio0_
    inner join
        Publication publicatio1_
            on publicatio0_.publicationId=publicatio1_.id
    where
        publicatio0_.authorId=?
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `InheritanceSingleTable` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

You can also map the entities of the inheritance hierarchy to multiple database tables. I show you how to do that in How to map an inheritance hierarchy to

multiple tables.

HIBERNATE SPECIFIC QUERIES AND MAPPINGS

Most of the Hibernate Tips in this book use features defined by the JPA specification. But Hibernate also provides a set of very useful extensions of the standard. The most commonly used ones are:

- How to join unassociated entities in a query
- How to map natural IDs
- How to load multiple entities by their primary key

If you’re using Hibernate 5, you should also have a look at the chapter about the Java 8 support in Hibernate. It shows some additional extensions to the JPA standard.

How to join unassociated entities in a query

Problem

JPQL requires a mapped association to join two entities. Is there any other option to join unassociated entities in a query?

Solution

Hibernate extends JPQL's limited join feature with the proprietary support for joins of unassociated entities. You can see an example of such a `JOIN` clause in the following code snippet.

```
Query q = em.createQuery("SELECT b.title, count(r.id) "
    + "FROM Book b JOIN Review r ON r.fkBook = b.id "
    + "GROUP BY b.title");
Object[] r = (Object[]) q.getSingleResult();
```

Hibernate uses the same syntax as you probably know from SQL. You reference the two entities you want to join and the kind of join you want to perform. Hibernate supports an inner `JOIN`, a `LEFT` outer join and a `RIGHT` outer join. You also need to define a join condition in the `ON` clause of the `JOIN` clause.

The previous code snippet shows an inner join of all `Book` and `Review` entities so that I can count the number of reviews for each book. When you execute this query, Hibernate transforms the JPQL `JOIN` clause into a similar looking SQL `JOIN` clause.

```
19:48:12,518 DEBUG [org.hibernate.SQL] -
    select
        book0_.title as col_0_0_,
        count(review1_.id) as col_1_0_
```

```
    from
        Book book0_
    inner join
        Review review1_
            on (
                review1_.fkBook=book0_.id
            )
    group by
        book0_.title
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `HibernateJoinUnassociatedEntities` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

How to map natural IDs

Problem

My domain model contains several natural IDs which I need to use to find objects in my business logic. What's the best way to models these IDs with Hibernate?

Solution

Hibernate provides proprietary support for natural IDs. It allows you to model them as a natural identifier of an entity and provides an additional API for retrieving them from the database.

The only thing you have to do to model an attribute as a natural id is to add the `@NaturalId` annotation to it. If your natural id consists of multiple attributes, you have to add this annotation to each of the attributes. You can see a simple example of such a mapping in the following code snippet. The ISBN number is a common natural id that is often used in the business logic to identify a book.

```
@Entity
public class Book {

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

    @NaturalId
    private String isbn;

    ...
}
```

Natural IDs are immutable by default. If you need mutable, natural identifier, you have to set the mutable attribute of the `@NaturalId` annotation to true.

Hibernate's `Session` interface provides the methods `byNaturalId` and `bySimpleNaturalId` to read an entity by its natural identifier from the database. The `byNaturalId` method allows you to use complex natural identifiers that consist of multiple attributes.

```
Book b = session.byNaturalId(Book.class)
                .using(Book_.isbn.getName(), "123-4567890123")
                .load();
```

You have to provide the class or the name of the entity as a parameter to the `byNaturalId` method to tell Hibernate which entity you want to retrieve. The following call of the `using` method provides the name of the natural ID attribute and its value. If the natural ID consists of multiple attributes, you have to call this method for each part of the ID. In this example, I use the JPA metamodel to reference the name of the `isbn` attribute in a type-safe way.

After you've provided the value of the natural id, you can call the `load`, `getReference` or `loadOptional` method to get the entity identified by it.

When your natural identifier consists of only one entity attribute, you can also use the `bySimpleNaturalId` method to load it. As you can see in the following code snippet, it provides a more convenient way to load entities with simple natural ids.

```
Book b = session.bySimpleNaturalId(Book.class)
                .load("123-4567890123");
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `HibernateNaturalId` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

How to load multiple entities by their primary key

Problem

I need to load multiple entities by their primary key. Is there an easier way to do that than writing a JPQL query?

Solution

Loading multiple entities by their primary keys is a very common use case. Since version 5.1, Hibernate offers a proprietary API that makes it a lot easier to load multiple entities and provides additional benefits, like fetching huge lists in multiple batches.

You just need to call the `byMultipleIds(Class entityClass)` method on the Hibernate `Session` and provide the class of the entities you want to load as a parameter. You then get a typed instance of the `MultiIdentifierLoadAccess` interface. You can use it to load multiple entities at once and to configure the database interaction.

```
MultiIdentifierLoadAccess<Book> multi =
    session.byMultipleIds(Book.class);
List<Book> books = multi.multiLoad(1L, 2L, 3L);
```

The call of the `multiLoad` method in this example loads the three `Book` entities with the given primary keys. Hibernate creates one query for this method call and provides the three primary keys as parameters to an IN statement.

```
06:25:30,202 DEBUG [org.hibernate.SQL] -
    select
        book0_.id as id1_0_0_,
```

```
        book0_.publishingDate as publishi2_0_0_,
        book0_.title as title3_0_0_,
        book0_.version as version4_0_0_
    from
        Book book0_
    where
        book0_.id in (
            ?,?,?
        )
```

Hibernate performs one or more queries to load the requested entities, when you provide more primary keys than the default batch size defined in your database-specific Hibernate dialect or the batch size you defined yourself. The following code snippet shows an example in which I call the `withBatchSize(int batchSize)` method to set the batch size to 2 and Hibernate has to perform two queries to select the three `Book` entities.

```
MultiIdentifierLoadAccess<Book> multi =
    session.byMultipleIds(Book.class);
List<Book> books = multi.withBatchSize(2).multiLoad(1L, 2L, 3L);
```

```
06:28:24,330 DEBUG [org.hibernate.SQL] -
    select
        book0_.id as id1_0_0_,
        book0_.publishingDate as publishi2_0_0_,
        book0_.title as title3_0_0_,
        book0_.version as version4_0_0_
    from
        Book book0_
    where
        book0_.id in (
            ?,?
        )
06:28:24,359 DEBUG [org.hibernate.SQL] -
    select
        book0_.id as id1_0_0_,
        book0_.publishingDate as publishi2_0_0_,
```

```
        book0_.title as title3_0_0_,
        book0_.version as version4_0_0_
    from
        Book book0_
    where
        book0_.id in (
            ?
        )
```

You can also tell Hibernate not to load any entities that are already stored in the first-level cache. That behavior is deactivated by default to avoid any overhead that could slow down your application. If you know that most of the requested entities are already in the cache, you can activate the additional check by calling the `enableSessionCheck` method.

```
MultiIdentifierLoadAccess<Book> multi =
    session.byMultipleIds(Book.class);
List<Book> books = multi.enableSessionCheck(true)
                        .multiLoad(1L, 2L, 3L);
```

When the `Book` entity with `id` 1 is already stored in the first-level cache, Hibernate doesn't add its primary key to the `IN` clause.

```
06:27:47,864 DEBUG [org.hibernate.SQL] -
    select
        book0_.id as id1_0_0_,
        book0_.publishingDate as publishi2_0_0_,
        book0_.title as title3_0_0_,
        book0_.version as version4_0_0_
    from
        Book book0_
    where
        book0_.id in (
            ?,?
        )
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `HibernateMultipleId` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

JAVA 8

Java 8 introduced a lot of powerful and convenient changes to the Java world. Hibernate supports the new concepts and data type since version 5. It started with the support of the classes of the Date and Time API as `BasicTypes` in version 5.0. The required changes were packaged into the optional `hibernate-java8.jar` file which you needed to add to the classpath of your application. Since version 5.1, this is no longer required. The Hibernate team switched the code base of their core project to Java 8 and started to use Java 8 in their APIs.

I show you how to use some of the most popular new data types and concepts in the following Hibernate tips:

- How to map an association to an Optional
- How to map classes of Java 8's Date and Time API
- How to retrieve a query result as a Java 8 Stream

How to map an association to an Optional

Problem

How do I map an optional to-one association to a Java 8 Optional?

Solution

Hibernate does not support Optional as an attribute type. But you can implement your own getter method to wrap the attribute in an `Optional<T>`, if Hibernate uses field access. This provides you the option to wrap the attribute which represents the to-one association into an Optional. You can see an example of it in the following code snippet.

```
@Entity
public class Book implements Serializable {

        ...

        @ManyToOne
        @JoinColumn(name="publisherid")
        private Publisher publisher;

        ...

        public Optional getPublisher() {
                return Optional.ofNullable(this.publisher);
        }

        public void setPublisher(final Publisher publisher) {
                this.publisher = publisher;
        }
}
```

As you can see in the code snippet, I wrap the `publisher` into an `Optional` in the `getPublisher` method and return an `Optional` instead of a `Publisher` entity. The caller of this method immediately sees that the `publisher` might be `null` and that she needs to handle it.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `MapOptionalAssociations` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Hibernate 5 also provides proprietary support for other Java 8 features, like `Streams` and the Date and Time API. You can read more about it in:

- How to retrieve a query result as a Java 8 Stream
- How to map classes of Java 8's Date and Time API

How to map classes of Java 8's Date and Time API

Problem

I want to use the Date and Time API in my domain model but I can't annotate them with JPA's `@Temporal` annotation. How do I use the new classes with Hibernate?

Solution

The good news is, that since Hibernate 5, you don't need any additional annotations to map classes of the Date and Time API. If you're using an older Hibernate version you need to implement your own mapping. I show you how to do that in How to map the Date and Time API with Hibernate 4.4.

Hibernate 5 supports the classes of the Date and Time API as `BasicType`. In contrast to the old `java.util.Date`, the classes of the Date and Time API provide all information Hibernate needs to map them to the correct JDBC types. The following table shows to which JDBC types Hibernate maps the new Java classes.

Table 1. JDBC Mappings

Java type	JDBC type
java.time.Duration	BIGINT
java.time.Instant	TIMESTAMP
java.time.LocalDateTime	TIMESTAMP
java.time.LocalDate	DATE

java.time.LocalTime	TIME
java.time.OffsetDateTime	TIMESTAMP
java.time.OffsetTime	TIME
java.time.ZonedDateTime	TIMESTAMP

As you can see in the following code snippet, you don’t need to provide any additional annotations when you use the Date and Time API classes as entity attribute types.

```
@Entity
public class Book {

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

    private LocalDate publishingDate;

    ...
}
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `DateAndTime` module of the example project. If you haven’t already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Hibernate 5 also provides proprietary support for other Java 8 features, like Optional and the Stream API. You can read more about it in:

- How to map an association to an Optional
- How to retrieve a query result as a Java 8 Stream

If you can’t use Hibernate 5 or if you’re not allowed to use any proprietary features, you can implement an AttributeConverter which converts the class of the Date and Time API to a supported Java type. I show you how to do that in How to map the Date and Time API with Hibernate 4.4.

How to retrieve a query result as a Java 8 Stream

Problem

I want to use a Java 8 Stream to process the result of a query. What is the best way to get a query result as a `Stream`?

Solution

The most obvious but not the most efficient approach is to just call the `stream` method on the `List` interface. You can see an example of it in the following code snippet. I first call the `getResultList` method on the `Query` interface to get the query result as a `List` and then call the `stream` method of the `List` interface.

```
List<Book> books = session.createQuery("SELECT b FROM Book b",
                                        Book.class).getResultList();
books.stream()
     .map(b -> b.getTitle() + " was published on "
            + b.getPublishingDate())
     .forEach(m -> log.info(m));
```

That approach might look OK but it has a drawback that can create performance issues for huge result sets. In this example, Hibernate gets all the selected `Book` entities from the database, stores them in memory, and puts them into a List. Then I call the `stream` method and process the results one by one.

There is no need to fetch all records of the query result at the beginning. For huge result sets it’s better to scroll through the records and fetch them in smaller chunks.

You might already know the concept from JDBC result sets or Hibernate's `ScrollableResults`. Scrolling through the records of a result set and processing them as a `Stream` are a great fit. Both approaches process one record after the other, and there is no need to fetch all of them upfront.

Since Hibernate 5.2, you can do exactly that with the `stream` method of Hibernate's `Query` interface. It returns a `Stream` of the query result and uses Hibernate's `ScrollableResults` internally. That allows you to scroll through the result set without fetching all records in the beginning.

```
Stream<Book> books = session.createQuery("SELECT b FROM Book b",
                                         Book.class).stream();
books.map(b -> b.getTitle() + " was published on "
              + b.getPublishingDate())
     .forEach(m -> log.info(m));
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `ResultsAsStreams` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Hibernate 5 also provides proprietary support for other Java 8 features, like Optional and the Date and Time API. You can read more about it in:

- How to map an association to an Optional
- How to map classes of Java 8's Date and Time API

LOGGING

Logging is an often ignored but important topic. A good logging configuration can help you to find potential issues during development and doesn't slow down your application in production.

The definition of a good production configuration is simple. You just need to set the log level of Hibernate's `org.hibernate` category to `ERROR` to avoid any unnecessary log messages.

The development configuration is more complex. You need more internal information while you implement new or change existing use cases. I show you how to get these information in the following Hibernate tips:

- How to log SQL statements and their parameters
- How to count the executed queries in a Session
- How to use query comments to identify a query

How to log SQL statements and their parameters

Problem

How do you configure Hibernate so that it writes the executed SQL statements and used bind parameters to the log file?

Solution

Hibernate uses two different log categories and log levels to log the executed SQL statements and their bind parameters:

- The SQL statements are written as `DEBUG` messages to the category `org.hibernate.SQL`.
- The bind parameter values are logged to the `org.hibernate.type.descriptor.sql` category with log level `TRACE`.

You can activate and deactivate them independently of each other in your log configuration.

> Logging all SQL queries and their bind parameter bindings can slow down your application and create huge log files. You shouldn't activate these log messages in production.

The following code snippet shows an example of a `log4j` configuration which activates both of them.

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{HH:mm:ss,SSS} %-5p
[%c] - %m%n

log4j.rootLogger=info, stdout
# basic log level for all messages
log4j.logger.org.hibernate=info

# SQL statements and parameters
log4j.logger.org.hibernate.SQL=debug
log4j.logger.org.hibernate.type.descriptor.sql=trace
```

Hibernate then writes log messages like the following ones to your log file.

```
17:34:50,353 DEBUG [org.hibernate.SQL] - select author0_.id as
id1_0_, author0_.firstName as firstNam2_0_, author0_.lastName as
lastName3_0_, author0_.version as version4_0_ from Author author0_
where author0_.id=1
17:34:50,362 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([id1_0_] : [BIGINT]) - [1]
17:34:50,373 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([firstNam2_0_] : [VARCHAR]) - [Thorben]
17:34:50,373 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([lastName3_0_] : [VARCHAR]) - [Janssen]
17:34:50,374 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([version4_0_] : [INTEGER]) - [0]
```

The SQL statement in the code snippet isn’t easy to read. That gets a lot better when you tell Hibernate to format it. You can do that by setting the configuration parameter `hibernate.format_sql` to true. You can provide

it as a system property or set it in the `persistence.xml` file, like in the following code snippet, or in the `hibernate.cfg.xml` file.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence>
    <persistence-unit name="my-persistence-unit">
        <description>Hibernate Tips</description>
        <provider>
            org.hibernate.jpa.HibernatePersistenceProvider
        </provider>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>

        <properties>
            <property name="hibernate.format_sql" value="true" />

            ...
        </properties>
    </persistence-unit>
</persistence>
```

The following code snippet shows the formatted SQL statement which is much better to read than the previous message.

```
16:42:56,873 DEBUG [org.hibernate.SQL] -
    select
        author0_.id as id1_0_,
        author0_.firstName as firstNam2_0_,
        author0_.lastName as lastName3_0_,
        author0_.version as version4_0_
    from
        Author author0_
    where
        author0_.id=?
16:42:56,926 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [1] as [BIGINT] - [1]
16:42:56,950 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([id1_0_] : [BIGINT]) - [1]
```

```
16:42:56,965 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([firstNam2_0_] : [VARCHAR]) - [Thorben]
```

```
16:42:56,965 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([lastName3_0_] : [VARCHAR]) - [Janssen]
16:42:56,966 TRACE
[org.hibernate.type.descriptor.sql.BasicExtractor] - extracted value
([version4_0_] : [INTEGER]) - [0]
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `LogSQLStatements` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

How to count the executed queries in a Session

Problem

Some of my use cases are slow and seem to perform too many queries. How do I count all queries executed within a Hibernate `Session`?

Solution

The easiest way to count all executed queries is to activate Hibernate's statistics component. Hibernate then collects a lot of internal statistics and provides them as a log message and via the Statistics API.

> Do not use this in production! Collecting the statistical information creates an overhead that slows down your application.

Hibernate's statistics component is deactivated by default. You can activate it by setting the `hibernate.generate_statistics` parameter to `true`. You can either do this by providing a system property with the same name or by setting the parameter in the `persistence.xml` file.

```
<persistence>
    <persistence-unit name="my-persistence-unit">
        <description>Hibernate Tips</description>
        <provider>
            org.hibernate.jpa.HibernatePersistenceProvider
        </provider>
        <properties>
            <property name="hibernate.generate_statistics"
                      value="true" />
            …
```

```
        </properties>
    </persistence-unit>
</persistence>
```

You have two options to access the statistics. Hibernate can write a subset with the most important information of each session to the log file or you can access them via the Statistics API.

Let's take a look at the log messages first. Hibernate writes a log message, similar to the following one, at the end of each session. It shows the number of SQL statements, the time spent for their preparation and execution and the interaction with the second-level cache.

```
16:24:55,318 INFO
[org.hibernate.engine.internal.StatisticalLoggingSessionEventListene
r] – Session Metrics {
25659 nanoseconds spent acquiring 1 JDBC connections;
22394 nanoseconds spent releasing 1 JDBC connections;
1091216 nanoseconds spent preparing 12 JDBC statements;
11118842 nanoseconds spent executing 12 JDBC statements;
0 nanoseconds spent executing 0 JDBC batches;
0 nanoseconds spent performing 0 L2C puts;
0 nanoseconds spent performing 0 L2C hits;
0 nanoseconds spent performing 0 L2C misses;
16999942 nanoseconds spent executing 1 flushes (flushing a total of
17 entities and 17 collections);
63915 nanoseconds spent executing 1 partial-flushes (flushing a
total of 0 entities and 0 collections)
```

You can also access the Statistics API via Hibernate's `Statistics` interface. You can get it from the `SessionFactory`. It provides several getter methods that give you access to more detailed information than the log output.

```
Statistics stats = sessionFactory.getStatistics();
long queryCount = stats.getQueryExecutionCount();
long collectionFetchCount = stats.getCollectionFetchCount();
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `CheckSQLStatementCount` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

If you want to learn more about Hibernate's logging features, you should have a look at these tips:

- How to log SQL statements and their parameters
- How to use query comments to identify a query

How to use query comments to identify a query

Problem

My application performs a lot of similar queries and I need to find the output of a specific query in my log file. Is there any way to make it easier to find a specific query in the log output?

Solution

Hibernate can add a comment when it generates an SQL statement for a JPQL or Criteria query or executes a native SQL query. You can see it in your application log file, when you activate SQL statement logging and in your database logs.

You need to activate SQL comments by setting the configuration parameter `hibernate.use_sql_comments` to true. The following code snippet shows an example configuration in the `persistence.xml` file.

```
<persistence>
        <persistence-unit name="my-persistence-unit">
                <description>Hibernate Tips</description>
                <provider>
                    org.hibernate.jpa.HibernatePersistenceProvider
        </provider>
                <exclude-unlisted-classes>false</exclude-unlisted-
classes>

                <properties>
                        <property name="hibernate.dialect"
                value="org.hibernate.dialect.PostgreSQLDialect" />
                        <property name="hibernate.use_sql_comments"
                            value="true" />
                        ....
```

```
                </properties>
        </persistence-unit>
</persistence>
```

When you activate SQL comments, Hibernate generates a comment for each query. These are often not useful to find a specific query. It's better to provide your own comment with the `org.hibernate.comment` query hint.

I use it in the following example to set the SQL comment for my query to "This is my comment".

```
TypedQuery q = em.createQuery(
    "SELECT a FROM Author a WHERE a.id = :id",
    Author.class);
q.setParameter("id", 1L);
q.setHint("org.hibernate.comment", "This is my comment");
Author a = q.getSingleResult();
```

Hibernate adds this comment to the generated SQL statement and writes it to the log file.

```
08:14:57,432 DEBUG [org.hibernate.SQL] – /* This is my comment */
select author0_.id as id1_0_, author0_.firstName as firstNam2_0_,
author0_.lastName as lastName3_0_, author0_.version as version4_0_
from Author author0_ where author0_.id=?
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `CommentSQLStatements` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Hibernate and JPA support several other hints. I summarized the most interesting ones in 11 JPA and Hibernate query hints every developer should know.

JPQL

JPQL is a query language defined by the JPA specification that you can use to read, update or delete entities. Its syntax is very similar to SQL but the queries are defined based on the entity object model instead of the table model. That makes it easier to use for most Java developers.

The following Hibernate tips show you several example use cases:

- How to create a JPQL query at runtime
- How to create a named JPQL query
- How to select a POJO with a JPQL query
- How to select multiple scalar values with a JPQL query
- How to initialize lazy relationships within a JPQL query
- How to downcast entities in JPQL queries
- How to call a standard function in a JPQL query
- How to call a user-defined function in a JPQL query
- How to use pagination with JPQL
- How to define a timeout for a JPQL query
- How to delete multiple entities with one JPQL query

- How to update multiple entities with one JPQL query

JPQL is comfortable to use but it also has two main disadvantages. The most noticeable one is that it is not as powerful as SQL. And it also introduces an additional abstraction on top of SQL and it’s not supported by relational databases. Hibernate needs to generate an SQL query based on the JPQL query to interact with the database.

How to create a JPQL query at runtime

Problem

I need to create a JPQL query based on user input. How do I do that at runtime?

Solution

You can call the `createQuery` method of the `EntityManager` with a JPQL query as a `String` to create an ad hoc query. The following code snippet shows an example of such a query. It selects the `Book` entity with a given id.

```
TypedQuery<Book> q = em.createQuery(
    "SELECT b FROM Book b WHERE b.id = :id", Book.class);
q.setParameter("id", 1L);
Book b = q.getSingleResult();
```

As you can see, I provide a `String` with the JPQL query and the class of the `Book` entity as parameters to the `createQuery` method. The JPQL syntax is pretty similar to SQL. But it uses the entity object model instead of database tables to define the query. That makes it very comfortable for us Java developers, but you have to keep in mind that the database still uses SQL. Hibernate, and any other JPA implementation has to transform the JPQL query into SQL. It is, therefore, a good practice to activate the logging of the SQL statements during development to check the generated SQL statements.

The query selects a `Book` entity with a given `id`, and I use the named bind parameter `:id` as a placeholder in the `WHERE` clause of the query. Bind parameters provide huge benefits compared to putting the parameter values directly into the query `String`. Hibernate maps the bind parameter value to the

correct type and escapes it if necessary. That makes them easier to handle and prevents SQL injection vulnerabilities. You can set the value of each bind parameter by calling the `setParameter` method of the `Query` or `TypedQuery` interface with the parameter name and its value.

I also provide the class of the `Book` entity as the second parameter to the `createQuery` method. This parameter is optional. It tells Hibernate the type of the query results and allows it to use the strongly typed `TypedQuery` instead of the untyped `Query` interface. You can see the main benefit of it in the last line of the code sample. The `getSingleResult` method of the `TypedQuery` interface returns a `Book` entity instead of an `Object`, and I don't need to cast the query result.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `JPQLAdHocQuery` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Ad-hoc queries are just one option to create a JPQL query. You can also use named queries that you statically define via annotations and reference by their name. I explain them in more detail in How to create a named JPQL query.

How to create a named JPQL query

Problem

I have queries that I want to use in multiple places in my business logic. What is the best way to define these queries once and use them when needed?

Solution

Named queries provide a good solution if you want to define a query once and use it in multiple parts of your business logic. You define a named query with the `@NamedQuery` annotation. The following code snippet shows a simple example of such a query definition.

```
@Entity
@NamedQuery(
    name = Book.QUERY_SELECT_BY_ID,
    query = "SELECT b FROM Book b WHERE b.id = :" + Book.PARAM_ID)
public class Book {

    public static final String QUERY_SELECT_BY_ID =
        "Book.selectById";
    public static final String PARAM_ID = "id";

    ...
}
```

The `@NamedQuery` annotation requires two parameters: The `name` and the JPQL `query` `String`. As you can see, I use a static `String` as the name of the query and the bind parameter. That makes it easier to reference them in the business logic.

The JPQL query is the same as you would use in an ad-hoc JPQL query and it looks very similar to an SQL query. The main difference between SQL and JPQL is that SQL uses database tables and JPQL the entity model to define the query. That makes JPQL easier to use for most Java developers, but you still need to be familiar with SQL. Hibernate transforms the JPQL statement to SQL, and you should always check the generated statements after you created or adapted a JPQL query.

That's all you need to do to define a named JPQL query. You can use its name to instantiate it in your business logic.

```
TypedQuery<Book> q = em.createNamedQuery(Book.QUERY_SELECT_BY_ID,
                                         Book.class);
q.setParameter(Book.PARAM_ID, 1L);
Book b = q.getSingleResult();
```

As you can see in the code snippet, you can use a named JPQL query in a similar way as an ad-hoc query. The only difference is the way you instantiate the `TypedQuery`. You already defined the query with the `@NamedQuery` annotation, and you only need to provide its `name` to the `createNamedQuery` method of the `EntityManager` to instantiate it. That is the point where the usage of the static `String` for the query name pays off. It's a lot easier to use the static `String` in your business code, and it allows you to refactor the query `name` easily.

I also provided the class of the `Book` entity as the second parameter to the `createNamedQuery` method to get a `TypedQuery` instance. This one is strongly typed and doesn't require any type casting of the query result.

The named query in this example selects a `Book` entity with a given `id`, and I used a named bind parameter in the `WHERE` clause of the query. I need to set its value before I can execute the query. I do that by calling the `setParameter` method of the `TypedQuery` method with the parameter name and its value.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `JPQLNamedQuery` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Named queries are just one option to create a JPQL query. You can also use ad-hoc queries to define queries at runtime. I explain them in more detail in How to create a JPQL query at runtime.

How to select a POJO with a JPQL query

Problem

The entity projection doesn't fit my use case. Is there an easy option to select POJOs instead of entities?

Solution

JPQL supports constructor expressions that allow you to define a constructor call in the `SELECT` clause of your query. You can see an example of such a query in the following code snippet.

```
TypedQuery<BookValue> q = em.createQuery(
    "SELECT new org.thoughts.on.java.model.BookValue("
    + "b.id, b.title, b.publisher.name) FROM Book b "
    + "WHERE b.id = :id", BookValue.class);
q.setParameter("id", 1L);
BookValue b = q.getSingleResult();
```

Instead of selecting an entity, I define a constructor expression in the `SELECT` clause. It consists of the keyword `new`, the fully qualified name of the class you want to instantiate and a comma separated list of parameters. The example in the previous code snippet calls the constructor of the `BookValue` class and provides the `id` and the `title` of the book and the `name` of the `Publisher` as parameters.

```
public class BookValue {

    public Long id;
    public String title;
    public String publisherName;
```

```
    public BookValue(Long id, String title, String publisherName) {
        this.id = id;
        this.title = title;
        this.publisherName = publisherName;
    }

    ...
}
```

Hibernate maps this statement to a SQL query that selects the required columns from the database and performs the defined constructor call for each record of the result set.

```
15:25:38,544 DEBUG [org.hibernate.SQL] -
    select
        book0_.id as col_0_0_,
        book0_.title as col_1_0_,
        publisher1_.name as col_2_0_
    from
        Book book0_,
        Publisher publisher1_
    where
        book0_.publisherid=publisher1_.id
        and book0_.id=?
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `JPQLConstructorExpression` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

POJOs are just one of the projections you can use with JPQL. You can also select managed entities or multiple scalar values:

- How to create a JPQL query at runtime
- How to select multiple scalar values with a JPQL query

How to select multiple scalar values with a JPQL query

Problem

I only need a few attributes of my entity. How do I select one or more scalar values in a JPQL query instead of entities?

Solution

You can select scalar values in the same way as you select managed entities. The following code snippet shows a JPQL query that selects the `title` of the `Book` entity and the `name` of the associated `Publisher` entity.

```
TypedQuery<Object[]> q = em.createQuery(
   "SELECT b.title, b.publisher.name FROM Book b WHERE b.id = :id",
   Object[].class);
q.setParameter("id", 1L);
Object[] result = q.getSingleResult();
```

As you can see, you can reference entity attributes instead of entities in the `SELECT` clause of your query. Hibernate maps this statement to a SQL query that only selects the required columns from the database and returns them as an `Object[]`. You can see the generated SQL query in the following code snippet.

```
15:37:16,708 DEBUG [org.hibernate.SQL] -
    select
        book0_.title as col_0_0_,
        publisher1_.name as col_1_0_
    from
        Book book0_,
        Publisher publisher1_
```

```
    where
        book0_.publisherid=publisher1_.id
        and book0_.id=?
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `JPQLScalarValues` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Scalar values are just one of the projections you can use with JPQL. You can also select managed entities or POJOs:

- How to create a JPQL query at runtime
- How to select a POJO with a JPQL query

How to initialize lazy relationships within a JPQL query

Problem

How do I initialize a lazy relationship within a query to avoid `LazyInitializationExceptions`?

Solution

Hibernate throws a `LazyInitializationException` if you try to use the attribute of a lazily fetched relationship outside of an active Hibernate `Session`.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

Author a = em.createQuery("SELECT a FROM Author a WHERE id = 1",
                          Author.class).getSingleResult();

log.info("Commit transaction and close Session");
em.getTransaction().commit();
em.close();

log.info(a.getFirstName()+" "+a.getLastName()+" wrote "
         +a.getBooks().size()+" book.");
```

```
06:23:38,969 DEBUG [org.hibernate.SQL] -
    select
        author0_.id as id1_0_,
        author0_.firstName as firstNam2_0_,
        author0_.lastName as lastName3_0_,
        author0_.version as version4_0_
    from
        Author author0_
    where
        author0_.id=1
```

```
06:23:38,993 INFO  [org.thoughts.on.java.model.TestJoinFetch] -
Commit transaction and close Session
06:23:39,008 ERROR [org.thoughts.on.java.model.TestJoinFetch] -
org.hibernate.LazyInitializationException: failed to lazily
initialize a collection of role:
org.thoughts.on.java.model.Author.books, could not initialize proxy
- no Session
```

You can avoid that by initializing the relationship before you close the `Session`. The easiest way to do that is a `JOIN FETCH` statement within a query, like the one in the following code snippet.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

Author a = em.createQuery(
    "SELECT a FROM Author a JOIN FETCH a.books WHERE a.id = 1",
    Author.class).getSingleResult();

em.getTransaction().commit();
em.close();

log.info(a.getFirstName()+" "+a.getLastName()+" wrote "
         +a.getBooks().size()+" book.");
```

The additional `FETCH` keyword tells Hibernate not only to join the entity for the query but also to fetch it from the database to initialize the attribute. That prevents `LazyInitializationExceptions` if you access the relationship attribute outside of an active Hibernate `Session`.

```
06:25:10,081 DEBUG [org.hibernate.SQL] -
    select
        author0_.id as id1_0_0_,
        book2_.id as id1_1_1_,
        author0_.firstName as firstNam2_0_0_,
        author0_.lastName as lastName3_0_0_,
        author0_.version as version4_0_0_,
```

```
        book2_.publisherid as publishe5_1_1_,
        book2_.publishingDate as publishi2_1_1_,
        book2_.title as title3_1_1_,
        book2_.version as version4_1_1_,
        books1_.authorId as authorId2_2_0__,
        books1_.bookId as bookId1_2_0__
    from
        Author author0_
    inner join
        BookAuthor books1_
            on author0_.id=books1_.authorId
    inner join
        Book book2_
            on books1_.bookId=book2_.id
    where
        author0_.id=1
06:25:10,128 INFO  [org.thoughts.on.java.model.TestJoinFetch] -
Commit transaction and close Session
06:25:10,147 INFO  [org.thoughts.on.java.model.TestJoinFetch] -
Thorben Janssen wrote 1 book.
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `JoinFetch` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

Join Fetch statements are only one option to initialize lazy relationships. Other interesting options are `@NamedEntityGraphs` and dynamic entity graphs, which allow you to define a query-independent graph of entities that is fetched with a query.

Initializing a required lazy relationship does not only prevent `LazyInitializationExceptions`, but it also can improve the

performance by avoiding n+1 select issues. You should always fetch the necessary relationships with the initial query. I explain the n+1 select issue in more detail and show you how to find and fix it in this free mini-course.

How to downcast entities in JPQL queries

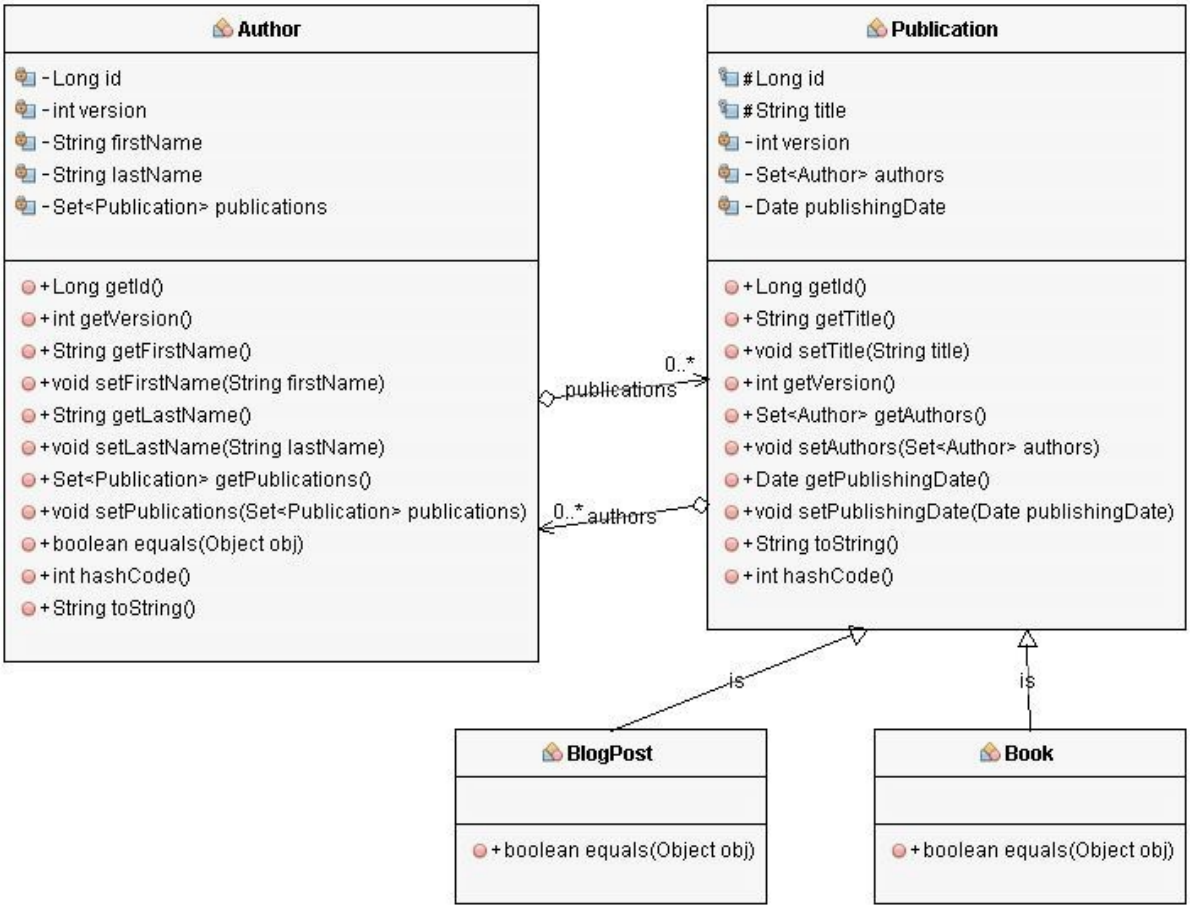
Problem

My domain model contains an inheritance structure, and I need to limit my query to a specific subclass. How do I do that with JPQL?

Solution

JPA 2.1 introduced the `TREAT` operator to JPQL, which you can use to downcast an entity within your query.

You can, for example, create a domain model with `Authors` who have written different kinds of `Publications`, like `Books` and `BlogPosts`. `Publication` is the super class of `Book` and `BlogPost` and you can model the relationship between the `Author` and the `Publication` entity.



You can now use the `TREAT` operator to downcast the `Publication` to `Book` and select all `Authors` who have written a `Book` about Java. The following code snippet shows an example of such a query.

```
List<Object[]> result = em.createQuery(
    "SELECT a, p FROM Author a JOIN a.publications p "
    + "WHERE treat(p AS Book).title LIKE '%Java%'")
                          .getResultList();
```

I use the inheritance strategy `SINGLE_TABLE` in this example, which maps all entities of the inheritance hierarchy to the `Publication` database table. As you can see in the generated SQL statement, Hibernate joins the records in the

`Author` table only with records in the `Publication` that have the value `Book` in the `DTYPE` column. These are the records that represent a `Book` entity.

```
06:10:59,866 DEBUG [org.hibernate.SQL] -
    select
        author0_.id as id1_0_0_,
        publicatio2_.id as id2_1_1_,
        author0_.firstName as firstNam2_0_0_,
        author0_.lastName as lastName3_0_0_,
        author0_.version as version4_0_0_,
        publicatio2_.publishingDate as publishi3_1_1_,
        publicatio2_.title as title4_1_1_,
        publicatio2_.version as version5_1_1_,
        publicatio2_.DTYPE as DTYPE1_1_1_
    from
        Author author0_
    inner join
        PublicationAuthor publicatio1_
            on author0_.id=publicatio1_.authorId
    inner join
        Publication publicatio2_
            on publicatio1_.publicationId=publicatio2_.id
            and publicatio2_.DTYPE='Book'
    where
        publicatio2_.title like '%Java%'
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `Treat` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

The `TREAT` operator is just one of several interesting new features introduced in JPA 2.1. You can get an overview of the different features and links to more detailed tutorials in JPA 2.1 – 12 features every developer should know.

The mapping of an inheritance structure is a complex task because relational table models provide no support for it. Hibernate and JPA support different strategies to map a hierarchy to one or more database tables.

You can learn more about it in How to map an inheritance hierarchy to one table and How to map an inheritance hierarchy to multiple tables.

How to call a standard function in a JPQL query

Problem

How do I call a database function in a JPQL query?

Solution

JPQL supports the following set of database functions that you can use in the `SELECT` and `WHERE` clause of your queries.

Table 1. JPQL Functions

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
`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
`treat(x as Type)`	Downcasts `x` to the given Type
`size(c)`	Returns the size of a given `Collection c`
`index(orderdCollection)`	Returns the index of the given value in an ordered `Collection`

The following code snippet shows a query that calls the `size` function on the `books` association.

```
Query q = em.createQuery(
    "SELECT a, size(a.books) FROM Author a GROUP BY a.id");
List<Object[]> results = q.getResultList();
```

The `size` function is JPA specific. You can use it to count the elements in a mapped association. As you can see in the log message, Hibernate generates a `JOIN` statement to join the associated table and calls the SQL `count` function to count the number of associated records in the `book` table.

```
05:47:23,682 DEBUG [org.hibernate.SQL] -
    select
        author0_.id as col_0_0_,
        count(books1_.authorId) as col_1_0_,
        author0_.id as id1_0_,
        author0_.firstName as firstNam2_0_,
        author0_.lastName as lastName3_0_,
        author0_.version as version4_0_
    from
```

```
        Author author0_ cross
    join
        BookAuthor books1_
    where
        author0_.id=books1_.authorId
    group by
        author0_.id
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `JpqlStandardFunction` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

JPQL supports only a subset of the functions supported by the SQL standard and no database-specific functions. Since JPA 2.1, you can use the function `function` to call functions unsupported by the JPA standard in a `CriteriaQuery` How to call a user-defined function in a CriteriaQuery.

How to call a user-defined function in a JPQL query

Problem

How do I call a user-defined database function in the WHERE clause of a JPQL query?

Solution

JPA 2.1 added the function `function(function_name {, function_arg})` to JPQL to provide a way to call user-defined and database-specific functions. You need to provide the name of the function as the first parameter and you can provide additional parameters that will be used as function arguments.

The following code snippet shows an example that calls the custom database function `calculate`. The function returns a `Double` and I provide the `price` of the `Book` and a query parameter as function arguments.

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

q.setParameter("double1", 10.0D);
q.setParameter("double2", 40.0D);
List<Book> books = q.getResultList();
```

You can use this approach in the `WHERE` clause to call all functions supported by your database.

You can also use the function `function` in the `SELECT` clause of your query. But you then need to register the database function so that Hibernate knows it's result type. This makes the function `function` superfluous because you can use all registered functions directly in your query.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `JPQLCustomFunction` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

You can also use the function `function` in a CriteriaQuery. I show you how in How to call a user-defined function in a CriteriaQuery.

How to use pagination with JPQL

Problem

JPQL does not support the `LIMIT` keyword. How do I use pagination with Hibernate?

Solution

With JPA and Hibernate, you have to set the pagination information on the `Query` interface and not in the query `String` as you would do it in SQL. You can do that by calling the `setFirstResult(int startPosition)` and `setMaxResults(int maxResults)` methods.

The following code snippet shows a simple example that returns the first five `Authors` from the database. The result set index is 0 based and you need to provide 0 as a `startPosition` to begin with the first element.

```
List<Author> authors = em.createQuery(
    "SELECT a FROM Author a ORDER BY a.id", Author.class)
                         .setMaxResults(5)
                         .setFirstResult(0)
                         .getResultList();
```

To select the next five `Authors` from the database, you only need to change the `startPosition` to 5.

```
List<Author> authors = em.createQuery(
    "SELECT a FROM Author a ORDER BY a.id", Author.class)
                         .setMaxResults(5)
                         .setFirstResult(5)
                         .getResultList();
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `QueryPagination` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

You can use the same approach to paginate the result of a `CriteriaQuery`. I explain it in more detail in How to use pagination with a CriteriaQuery.

How to define a timeout for a JPQL query

Problem

I want to make sure that my query gets canceled after a certain time. Is there a way to define a timeout for queries?

Solution

JPA and Hibernate support the `javax.persistence.query.timeout` query hint to define a query timeout in milliseconds. Hibernate uses it to call the `setTimeout` method on the JDBC `Statement` and doesn't handle the timeout itself. It, therefore, depends on the JDBC driver and the database capabilities, if the query hint has any effect.

The following two code snippets show you how to provide the query timeout hint to a `Query` and the `EntityManager.find` method.

```
List<Author> authors = em.createQuery("SELECT a FROM Author a")
                  .setHint("javax.persistence.query.timeout", 1)
                  .getResultList();
```

```
HashMap<String, Object> hints = new HashMap<>();
hints.put("javax.persistence.query.timeout", 1);

em.find(Author.class, 1L, hints);
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `QueryTimeout` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

The query timeout is just one of multiple query hints supported by JPA and Hibernate. I explain more of them in 11 JPA and Hibernate query hints every developer should know.

How to delete multiple entities with one JPQL query

Problem

I need to delete multiple entities. Is there a way to do that with one JPQL query like in SQL?

Solution

You can use a JPQL `DELETE` statement to remove multiple entities at once. It's syntax is similar to an SQL `DELETE` statement. You can see an example of a JPQL statement that deletes all records from the `Book` table in the following code snippet.

```
Query query = em.createQuery("DELETE Book b");
query.executeUpdate();
```

You just need to call the `createQuery` method of the `EntityManager` with a JPQL `DELETE` statement. The JPQL syntax looks very similar to the SQL syntax. It starts with the keyword `DELETE` and a reference to the kind of entity you want to delete. You can also add an optional `WHERE` clause, like you use in a JPQL `SELECT` statement, if you just want to delete a specific set of entities. After you defined the query, you can call the `setParameter` method of the `Query` interface to set bind parameter values and execute the query by calling the `executeUpdate` method. I didn't use any bind parameters in my `DELETE` statement and, therefore, skip the call of the `setParameter` method.

Hibernate generates a SQL `DELETE` statement based on your JPQL statement. In this example, it also manages the mapped associations. It first performs an

SQL `DELETE` statement to remove all references to the deleted `Book` entities from the association table `BookAuthor`. When that is done, Hibernate executes another SQL statement to remove the `Book` entities.

```
17:42:55,154 DEBUG [org.hibernate.SQL] - delete from BookAuthor
where (bookId) in (select id from Book)
17:42:55,155 DEBUG [org.hibernate.SQL] - delete from Book
```

Deleting multiple entities with a JPQL `DELETE` statement is more efficient than removing them one by one. But Hibernate doesn't know which database records the statement deletes and doesn't remove any entities from the first-level cache. You need to make sure that the cache doesn't contain any deleted entities or you need to invalidate the cache programmatically.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `JPQLDelete` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

A JPQL `DELETE` statement is just one option to delete multiple entities with one query. You can also use the `CriteriaDelete` statement as I explain in How to delete multiple entities with the Criteria API

How to update multiple entities with one JPQL query

Problem

I have to update multiple entities. With SQL, I can do that with one `UPDATE` statement. Is there a way to do the same with a JPQL query?

Solution

You can create a JPQL `UPDATE` statement with a similar syntax as you know from SQL. The following code snippet shows an example of such a statement. It increases the price of all `Books` by 10%.

```
Query query = em.createQuery(
    "UPDATE Book b SET b.price = b.price*1.1");
query.executeUpdate();
```

As you can see, the JPQL `UPDATE` statement looks pretty similar to an SQL `UPDATE` statement. I first define which entity I want to update and define the update operation in the `SET` clause.

This statement updates all `Book` entities. If you only want to update a specific set of entities, you can add a `WHERE` clause to the statement. You can define it in the same way as you do it for a JPQL `SELECT` statement.

You can check the generated SQL statement in the log file if you activate logging for the SQL statements.

```
18:57:25,118 DEBUG [org.hibernate.SQL] - update Book set
price=price*1.1
```

Updating multiple entities with a JPQL `UPDATE` statement is more efficient than updating the entities one by one. But Hibernate doesn't know which database records get changed and doesn't update any entities in the first-level cache. You need to make sure that the cache doesn't contain any entities affected by the update statement or you need to invalidate the cache programmatically.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `JPQLUpdate` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

A JPQL `UPDATE` statement is just one option to update multiple entities with 1 query. You can create the same statement with the Criteria API as I explain in How to update multiple entities with the Criteria API.

NATIVE SQL QUERIES

JPQL is the preferred query language of most Hibernate users because it operates on the entity model and the defined associations instead of the table model. But it implements only a small subset of the SQL feature set and that is most often not enough to implement all queries of complex business applications.

That is the point when native SQL queries come into play. JPA and Hibernate are designed as a leaky abstraction and allow you to implement and execute native SQL queries within your current context. The following Hibernate tips show you how to do that:

- How to create a native SQL query at runtime
- How to create a named native SQL query
- How to map the result of a native SQL query to entities
- How to map the result of a native SQL query to a POJO

How to create a native SQL query at runtime

Problem

I need to adapt a native query based on user input. How do I create a native SQL query at runtime?

Solution

You can create ad-hoc native queries in a similar way as you create ad-hoc JPQL queries. You just need to provide a `String` with the SQL statement to the `createNativeQuery` method of the `EntityManager`.

I do that in the following code snippet with an SQL query that selects a record from the `book` table with an `id` equal to a bind parameter value. The `?` you can see at the end of the query is the placeholder for a positional bind parameter. You can set the bind parameter values of a native query in the same way as you set the bind parameters of a JPQL query. You just need to call the `setParameter` method of the `Query` interface with the parameter position and its value.

> The index of positional parameters in ad-hoc native queries starts at 1.

I also provide the class of the `Book` entity as an additional parameter to tell Hibernate to map the query result to a `Book` entity. I explain the mapping of native query results to entities in more detail in How to map the result of a native SQL query to entities.

```
Query q = em.createNativeQuery("SELECT * FROM book b WHERE id = ?",
                               Book.class);
q.setParameter(1, 1);
Book b = (Book) q.getSingleResult();
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `NativeQuery` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Native queries don't return the strongly typed results you know from JPQL queries. They return an `Object[]` or a `List<Object[]>` and you need to cast them yourself or tell Hibernate how to handle the result. Hibernate can map the query result to entities or POJOs, as I show you in:

- How to map the result of a native SQL query to entities
- How to map the result of a native SQL query to a POJO

How to create a named native SQL query

Problem

I don’t want to define the native query in my business logic. Is there an option to specify a named native query similar to JPQL?

Solution

You can define named native SQL queries with the `@NamedNativeQuery` annotation. As you can see in the following code snippet, it’s very similar to the `@NamedQuery` annotation you use to define a named JPQL query.

```
@Entity
@NamedNativeQuery(name=Book.QUERY_SELECT_BY_ID,
                  query="SELECT * FROM book b WHERE id = ?",
                  resultClass = Book.class)
public class Book {

    public static final String QUERY_SELECT_BY_ID =
        "Book.selectById";

    ...
}
```

You need to provide a `name` and a native SQL `query`. As you can see, I use a static `String` as the name of the query. That makes it easier to reference it in the business logic. The SQL query in this example is very simple. It uses a positional bind parameter to select a record from the `book` table. Hibernate also supports named bind parameters for native SQL queries but the JPA specification does not. So, you should use positional bind parameters, if you want to be able to use your application with a different JPA implementation.

I also provide a `resultClass` in this example to tell Hibernate to map the query result to a `Book` entity. That's all you need to do to define a named native query.

You can use this query in the same way as a named JPQL query.

```
Query q = em.createNamedQuery(Book.QUERY_SELECT_BY_ID);
q.setParameter(1, 100);
Book b = (Book) q.getSingleResult();
```

You instantiate a new `Query` by calling the `createNamedQuery` method of the `EntityManager` with the name of the `@NamedNativeQuery`. That is the same `Query` interface as you know from your JPQL queries. You can use it to set bind parameter values or define pagination. In this example, I just call the `setParameter` method to set the value for the positional bind parameter.

> The index of positional parameters in named native queries starts at 1.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `NamedNativeQuery` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

Native queries don't return the strongly typed results you know from JPQL queries. They return an `Object[]` or a `List<Object[]>` and you need to

cast them yourself or tell Hibernate how to handle the result. Hibernate can map the query result to entities or POJOs, as I show you in:

- How to map the result of a native SQL query to entities
- How to map the result of a native SQL query to a POJO

How to map the result of a native SQL query to entities

Problem

My query is too complex for JPQL, and I have to use a native query. Is there a way to map the result of the query to managed entities?

Solution

If your query returns all columns that are mapped to an entity, you can tell Hibernate to map the result to a managed entity. Afterward, you can use the entity in the same way as any other managed entity.

There are two options to define the mapping:

1. You can use an implicit mapping if your query result uses the same column names as your entity mapping.
2. You can create a custom mapping if the column names do not match the entity mapping.

Implicit Mapping

If you can use it, the implicit mapping is the easier to use and better approach for most use cases. You only need to provide the class of the entity as the second parameter to the `createNativeQuery` method.

```
Book b = (Book) em.createNativeQuery(
    "SELECT * FROM book b WHERE id = 1", Book.class)
                .getSingleResult();
```

Explicit Mapping

If the column names of your query result do not match the column names of your entity mapping, you have to define the mapping yourself. The following query shows a very simple example of such a situation. It renames the column `id` to `bookId` so that Hibernate can not use the implicit mapping.

```
Book b = (Book) em.createNativeQuery(
    "SELECT id as bookId, version, title, "
    + "publishingDate, publisherid "
    + "FROM book b WHERE id = 1", "BookMapping")
                .getSingleResult();
```

You can define a custom result mapping with a `@SqlResultSetMapping` annotation. It specifies the entity class and a mapping for each entity attribute.

```
@SqlResultSetMapping(
    name = "BookMapping",
    entities = @EntityResult(
        entityClass = Book.class,
        fields = {
            @FieldResult(name = "id", column = "bookId"),
            @FieldResult(name = "version", column = "version"),
            @FieldResult(name = "title", column = "title"),
            @FieldResult(name = "publishingDate",
                         column = "publishingDate"),
            @FieldResult(name = "publisher",
                         column = "publisherid")}))
```

As you can see in the code snippet, I set a `name` and a `@EntityResult` annotation on the `@SqlResultSetMapping` annotation. To use the mapping, you need to provide its `name` as the second parameter to the `createNativeQuery` method. So, make sure to choose a name that describes the mapping and is easy to remember.

The `@EntityResult` annotation defines to which entity the query result will be mapped. Therefore, you need to specify the class of the entity and a set of `@FieldResult` annotations. Each `@FieldResult` annotation defines the mapping between a result set column and an entity attribute. You can define multiple `@EntityResult` mappings for a `@SqlResultSetMapping` if you want to map the query result to multiple entities.

You can then use this mapping by providing its name as the second parameter to the `createNativeQuery` method. The defined `@SqlResultSetMapping` tells Hibernate to map the query result to a `Book` entity, but it doesn't change the return type of the `getSingleResult` method. It still returns an `Object` and you need to cast it to a `Book` entity.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `MapNativeQueryToEntity` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

`@SqlResultSetMapping` is a powerful feature that allows you to define complex mappings for native query results. You can also use it to map the query result to a POJO as I show in How to map the result of a native SQL query to a POJO.

How to map the result of a native SQL query to a POJO

Problem

My query is too complex for JPQL, and I have to use a native query. What is the easiest way to map the result of the query to a POJO?

Solution

JPA supports `@SqlResultSetMappings`, which you can use to map the query result to a POJO. In the following example, I want to map the result of a native SQL query to `BookValue` objects.

```
public class BookValue {

    private String title;
    private Date publishingDate;

    public BookValue(String title, Date publishingDate) {
        this.title = title;
        this.publishingDate = publishingDate;
    }

    ...
}
```

You can use a `@SqlResultSetMapping` to define a constructor call similar to the constructors expressions you know from JPQL. The following code snippet shows an example of such a mapping.

```
@SqlResultSetMapping(
    name = "BookValueMapping",
    classes = @ConstructorResult(
```

```
        targetClass = BookValue.class,
        columns = {@ColumnResult(name = "title"),
                   @ColumnResult(name = "date")}))
```

The mapping consist of a `name` and a `@ConstructorResult` annotation. To use the mapping, you need to provide its `name` as the second parameter to the `createNativeQuery` method. So, make sure to choose a name that describes the mapping and is easy to remember.

The `@ConstructorResult` annotation defines a constructor call of the `BookValue` class. The `@ColumnResult` annotations describe how Hibernate shall map the columns of the result to the parameters of the constructor. In this example, Hibernate performs a constructor call using the value of the `title` column as the first and the value of the `date` column as the second parameter.

When you provide the name of the `@SqlResultSetMapping` as the second parameter to the `createNativeQuery` method, Hibernate applies the mapping to the query result. You can use it to map the results of all queries that return at least the parameters defined by the `@ColumnResult` annotations.

```
BookValue b = (BookValue) em.createNativeQuery(
    "SELECT b.publishingDate as date, b.title, b.id "
    + "FROM book b WHERE b.id = 1", "BookValueMapping")
                            .getSingleResult();
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `MapNativeQueryToPojo` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

`@SqlResultSetMapping` is a powerful feature that allows you to define complex mappings for native query results. You can also use it to map the query result to multiple POJOs or managed entities as I show in How to map the result of a native SQL query to entities.

CREATE QUERIES PROGRAMMATICALLY WITH THE CRITERIA API

Like all JPA implementations, Hibernate implements the Criteria API. It supports the same feature set as JPQL and is a good solution if you need to dynamically define queries at runtime. Common use cases for that are complex search operations based on user input. For these use cases, the dynamic approach of the Criteria API is often a better solution than defining multiple JPQL queries.

But as you'll see in the examples, the definition of a query can become complex. I therefore recommend using JPQL for all use cases that allow the static definition of a query.

This chapter includes the following Hibernate tips:

- How to select entities with a CriteriaQuery
- How to select POJOs with a CriteriaQuery
- How to select multiple scalar values in a CriteriaQuery
- How to call a standard database function in a CriteriaQuery
- How to call a user-defined function in a CriteriaQuery
- How to update multiple entities with the Criteria API

- How to delete multiple entities with the Criteria API
- How to use pagination with a CriteriaQuery
- How to reference entity attributes in a type-safe way

How to select entities with a CriteriaQuery

Problem

I need to define a query based on user input. What is the best way to programmatically create a query at runtime with Hibernate?

Solution

You can use JPA's Criteria API to define queries programmatically. It provides a type-safe way to create a query and supports the same features as JPQL.

> Hibernate's proprietary Criteria API is deprecated. You should use JPA's Criteria API instead.

The following code snippet shows the definition of a `CriteriaQuery`, which selects all `Books` written by an `Author` with a given first and last name.

You first need to get a `CriteriaBuilder` instance from the `EntityManager`. It's a factory class that helps you to define different parts of your query, like bind parameters, function calls, and predicates.

You then create a `CriteriaQuery` instance that is the root of the object graph that represents your query. I recommend providing the return type of your query as a parameter to the `createQuery` method. It creates a typed instance of the `CriteriaQuery` interface. I want to select `Book` entities in this example and, therefore, provide `Book.class` as the parameter.

In the next step, you define the `FROM` clause of the query. In this example, I use the `Book` entity as the root of the query. Then I call the join method on the `Root` interface to join the `Book` entity with the `Author` entity. I use the JPA Metamodel class `Book_` to reference the `authors` attribute of the `Book` entity. If you're not familiar with the JPA Metamodel, take a look at How to reference entity attributes in a type-safe way, which provides a comfortable and type-safe way to reference entity attributes.

The definition of the `WHERE` clause is optional. If you want to select all records from the database, you can skip this block and execute your query. I only want to get the `Books` written by `Authors` with a given first and last name and define a `WHERE` clause for my query.

The first thing you should do is define the bind parameters you want to use in your query. I create two parameters of type `String`. One for the first name and one for the last name. Then I define the `WHERE` clause by calling the `where` method on the `CriteriaQuery` interface with a `Predicate`. The `Predicate` in this example checks that the `firstName` and the `lastName` attribute are equal to the values of the bind parameters.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Book> cq = cb.createQuery(Book.class);

// define FROM clause
Root<Book> root = cq.from(Book.class);
SetJoin<Book, Author> authors = root.join(Book_.authors);

// define WHERE clause with bind parameters
ParameterExpression<String> paramFirstName =
    cb.parameter(String.class);
ParameterExpression<String> paramLastName =
    cb.parameter(String.class);
```

```
cq.where(
    cb.and(
        cb.equal(authors.get(Author_.firstName), paramFirstName),
        cb.equal(authors.get(Author_.lastName), paramLastName)));

// execute query
TypedQuery<Book> query = em.createQuery(cq);
query.setParameter(paramFirstName, "Thorben");
query.setParameter(paramLastName, "Janssen");
List<Book> books = query.getResultList();
```

I've defined my query, and now I want to execute it. That requires multiple steps, as you can see in the code sample.

You first need to call the `createQuery` method of the `EntityManager` with your `CriteriaQuery` instance. Hibernate returns an instance of the `TypedQuery` interface. That is the same interface as you get when you create a JPQL query and you're probably familiar with the following steps. I set the values for the two bind parameters by calling the `setParameter` method for each of them and call the `getResultList` method on the `TypedQuery` interface to execute the query.

Hibernate then generates a SQL query based on the `CriteriaQuery` and executes it with the provided bind parameter values.

```
13:43:58,566 DEBUG [org.hibernate.SQL] -
    select
        book0_.id as id1_1_,
        book0_.price as price2_1_,
        book0_.publisherid as publishe6_1_,
        book0_.publishingDate as publishi3_1_,
        book0_.title as title4_1_,
        book0_.version as version5_1_
    from
        Book book0_
```

```
    inner join
        BookAuthor authors1_
            on book0_.id=authors1_.bookId
    inner join
        Author author2_
            on authors1_.authorId=author2_.id
    where
        author2_.firstName=?
        and author2_.lastName=?
13:43:58,571 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [1] as [VARCHAR] - [Thorben]
13:43:58,572 TRACE [org.hibernate.type.descriptor.sql.BasicBinder] -
binding parameter [2] as [VARCHAR] - [Janssen]
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `CriteriaQuery` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

The `CriteriaQuery` in this example selects only one entity class. That is the most common projection. But you can also use POJOs or more complex projections that select multiple entities, POJOs or scalar values. I provide examples for that in:

- How to select POJOs with a CriteriaQuery
- How to select multiple scalar values in a CriteriaQuery

How to select POJOs with a CriteriaQuery

Problem

JPQL supports constructor expressions to select POJOs instead of entities or scalar values. Can I do the same with the Criteria API?

Solution

You can use a similar constructor expression with the Criteria API as you use in JPQL queries. In the following example, I want to select `AuthorValue` objects.

```
public class AuthorValue {

    private String firstName;
    private String lastName;

    public AuthorValue(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    ...
}
```

The definition of the `CriteriaQuery` follows the same approach as you use to select entities.

You first need to get a `CriteriaBuilder` instance from the `EntityManager`. It's a factory class that helps you to define different parts of your query, like bind parameters, function calls, and predicates.

You then create a `CriteriaQuery` instance that is the root of the object graph that represents your query. I recommend providing the return type of your query as a parameter to the `createQuery` method. It creates a typed instance of the `CriteriaQuery` interface. I want to select `AuthorValue` objects in this example and, therefore, provide `AuthorValue.class` as the parameter.

In the next step, you define the `FROM` clause of the query. In this example, I use the `Author` entity as the root of the query.

Then you can define the projection of your query. In this example, it's a call of the `AuthorValue` constructor. The `construct` method of the `CriteriaBuilder` allows you to define constructor calls. Call this method using the class that Hibernate instantiates as the first parameter and an optional list of `Selections` that are used as constructor parameters.

In this example, I use the JPA Metamodel class `Author_` to reference the `firstName` and `lastName` attributes of the `Author` entity. If you're not familiar with the JPA Metamodel, take a look at How to reference entity attributes in a type-safe way, which provides a comfortable and type-safe way to reference entity attributes.

The definition of the `WHERE` clause is optional. If you want to select all records from the database, you can skip this block and execute your query. That's what I do in this example. You can see an example of a `WHERE` clause definition in How to select entities with a CriteriaQuery.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<AuthorValue> q = cb.createQuery(AuthorValue.class);
Root<Author> root = q.from(Author.class);
q.select(
```

```
    cb.construct(
        AuthorValue.class,
        root.get(Author_.firstName),
        root.get(Author_.lastName)));
```

That's all you need to do to define the `CriteriaQuery`. You can now execute it in two steps. You first need to call the `createQuery` method of the `EntityManager` with your `CriteriaQuery`. This method call returns the same `TypedQuery` interface as you use in your JPQL queries. You can use it to set bind parameter values or to paginate the query result. I don't use any bind parameters in this example and therefore, can skip this part. In the final step, you need to call the `getResultList` method on the `TypedQuery` interface to execute the query and retrieve a `List` of `AuthorValue` objects.

```
TypedQuery<AuthorValue> query = em.createQuery(q);
List<AuthorValue> authors = query.getResultList();
```

When I created the query, I referenced the `firstName` and `lastName` attributes in the constructor call definition. As you can see in the following log messages, Hibernate used these reference to generates an SQL query that selects the `firstName` and `lastName` columns from the `Author` table.

```
13:43:09,884 DEBUG [org.hibernate.SQL] -
    select
        author0_.firstName as col_0_0_,
        author0_.lastName as col_1_0_
    from
        Author author0_
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `CriteriaConstructor` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

If you don't want to create a POJO to represent your query result, you can also select a set of scalar values. I show you how to do that in How to select multiple scalar values in a CriteriaQuery.

And if you want to use the Criteria API in your project, you should also have a look at the JPA metamodel. It provides a great way to create queries in a type-safe way. I show you how to use and generate the required classes in How to reference entity attributes in a type-safe way.

How to select multiple scalar values in a CriteriaQuery

Problem

How do I select a list of scalar values in a Criteria query?

Solution

You can select multiple scalar values using the `multiselect` method of the `CriteriaQuery` interface. The following code snippet shows an example of such a query.

```
// Prepare query
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Author> author = q.from(Author.class);

// Select multiple scalar values
q.multiselect(
    author.get(Author_.firstName).alias("firstName"),
    author.get(Author_.lastName).alias("lastName"));
```

The definition of the `CriteriaQuery` follows the same approach as you use to select entities.

You first need to get a `CriteriaBuilder` instance from the `EntityManager`. It's a factory class that helps you to define different parts of your query, like bind parameters, function calls, and predicates.

You then create a `CriteriaQuery` instance by calling the `createTupleQuery` method of the `CriteriaBuilder`. The `CriteriaQuery` is the root of the object graph that represents your query.

In the next step, you define the `FROM` clause of the query. In this example, I use the `Author` entity as the root of the query.

After that is done, you can define the projection of your query by calling the `multiselect` method. It expects a `List` or an array of `Selection` interfaces that define the entity attributes you want to fetch from the database. In this example, I use the JPA Metamodel to reference the attribute `firstName` and `lastName` in a type-safe way.

That's all you need to do to define the `CriteriaQuery`. You can now execute the query in two steps. You first need to call the `createQuery` method of the `EntityManager` with your `CriteriaQuery` to get an instance of the `TypedQuery` interface. This is the same interface as you use for your JPQL queries and you can use it to set bind parameter values or to paginate the query result. You can the execute the query by calling the `getResultList` method on the `TypedQuery` interface.

The `getResultList` method returns a `List` of `Tuple` interface implementations. The `Tuple` interface provides convenient access to the selected values based on their position or alias. In the code snippet, I defined an alias for each attribute in the query and use it to get them from the `Tuple` result.

```
TypedQuery<Tuple> query = em.createQuery(q);
List<Tuple> authorNames = query.getResultList();

for (Tuple authorName : authorNames) {
    log.info(authorName.get("firstName")
        + " "
        + authorName.get("lastName"));
}
```

As you can see in the log messages, Hibernate generated a SQL `SELECT` statement for the `CriteriaQuery`. It selects the columns that are mapped by the `firstName` and `lastName` attributes of the `Author` entity.

```
14:19:39,392 DEBUG [org.hibernate.SQL] -
    select
        author0_.firstName as col_0_0_,
        author0_.lastName as col_1_0_
    from
        Author author0_
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `CriteriaTuples` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

If you want to select a POJO instead of multiple scalar values, you can define a constructor call with the `construct` method of the `CriteriaBuilder`. I explain it in more detail in How to select POJOs with a CriteriaQuery.

And if you want to use the Criteria API in your project, you should also have a look at the JPA metamodel. It provides a great way to create queries in a type-safe way. I show you how to use and generate the required classes in How to reference entity attributes in a type-safe way.

How to call a standard database function in a CriteriaQuery

Problem

How do I call a database function in a CriteriaQuery?

Solution

The Criteria API supports a set of database functions you can use in your queries. You can define function calls with the `CriteriaBuilder`. It provides a method for each supported function.

Table 1. Functions

CriteriaBuilder method	Description
`upper(Expression<String> x)`	Create an `Expression` to transform `String x` to upper case
`lower(Expression<String> x)`	Create an `Expression` to transform `String x` to lower case
`currentDate()`	Create an `Expression` to return the current date of the database
`currentTime()`	Create an `Expression` to return the current time of the database
`currentTimestamp()`	Create an `Expression` to return a timestamp of the current date and time of the database
`substring(Expression<String> s, Expression<Integer> from)`	Create an `Expression` to return a substring of the given `String`

`substring(Expression<String> s, Expression<Integer> from, Expression<Integer> len)`	Create an `Expression` to return a substring of the given `String`
`substring(Expression<String> s, int from)`	Create an `Expression` to return a substring of the given `String`
`substring(Expression<String> s, int from, int len)`	Create an `Expression` to return a substring of the given `String`
`trim(char t, Expression<String> x)`	Create an `Expression` to remove leading and trailing character from the given `String`
`trim(CriteriaBuilder.Trimspec ts, char t, Expression<String> x)`	Create an `Expression` to remove character from the given `String`
`trim(CriteriaBuilder.Trimspec ts, Expression<String> x)`	Create an `Expression` to remove whitespaces from the given `String`
`trim(Expression<Character> t, Expression<String> x)`	Create an `Expression` to remove leading and trailing character from the given `String`
`trim(Expression<String> x)`	Create an `Expression` to remove leading and trailing whitespaces from the given `String`
`length(Expression<String> x)`	Create an `Expression` to return the length of the given `String x`
`locate(Expression<String> x, Expression<String> pattern)`	Create an `Expression` to return the position of the `String` pattern in `x`.
`locate(Expression<String> x, Expression<String> pattern, Expression<Integer> from)`	Create an `Expression` to return the position of the `String` pattern in `x`. The search starts at the position `from`
`locate(Expression<String> x, String pattern, Expression<Integer> from)`	Create an `Expression` to return the position of the `String pattern` in `x`. The search starts at

	the position `from`
`locate(Expression<String> x, String pattern, int from)`	Create an `Expression` to return the position of the `String pattern` in `x`. The search starts at the position `from`
`abs(Expression<N> x)`	Create an `Expression` to return the absolute value of the given `Expression`
`avg(Expression<N> x)`	Create an `Expression` to return the average of the given `Expression`
`sqrt(Expression<? extends Numeric> x)`	Create an `Expression` to return the square root of the given number
`mod(Expression<Integer> x, Expression<Integer> y)`	Create an `Expression` to return the remainder of a division
`mod(Expression<Integer> x, Integer y)`	Create an `Expression` to return the remainder of a division
`mod(Integer x, Integer y)`	Create an `Expression` to return the remainder of a division
`treat(CollectionJoin<X,T> join, Class<E> type)`	Create an `Expression` to downcast a `CollectionJoin` to the given type
`treat(Join<X,T> join, Class<V> type)`	Create an `Expression` to downcast a `Join` to the given type
`treat(ListJoin<X,T> join, Class<E> type)`	Create an `Expression` to downcast a `ListJoin` to the given type
`treat(MapJoin<X,K,T> join, Class<V> type)`	Create an `Expression` to downcast a `MapJoin` to the given type

`treat(Path<X> path, Class<T> type)`	Create an `Expression` to downcast a `Path` to the given type
`treat(Root<X> root, Class<T> type)`	Create an `Expression` to downcast a `Root` to the given type
`treat(SetJoin<X,T> join, Class<E> type)`	Create an `Expression` to downcast a `SetJoin` to the given type
`size(C collection)`	Create an `Expression` to return the size of a given collection
`size(Expression<C> collection)`	Create an `Expression` to return the size of a given collection

The following code snippet shows a query that calls the `size` function on the `books` association.

I first call the `getCriteriaBuilder` method on the `EntityManager` to get a `CriteriaBuilder` instance. Then I create a `CriteriaQuery` that selects a `Tuple`. I get into more detail about this kind of projection in How to select multiple scalar values in a CriteriaQuery.

The definition of the projection is the important part of this code sample. As you can see in the code, I select the `Author` entity and call the `size` function to count the number of `Books` the `Author` has written. The `size` function also requires me to use a `GROUP BY` clause on the primary key of the `Author` entity.

That's all you need to do to call a function in a `CriteriaQuery`. You can now use it to create a `TypedQuery` and execute it.

```
// Define the CriteriaQuery
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<Author> root = cq.from(Author.class);
cq.multiselect(root, cb.size(root.get(Author_.books)));
cq.groupBy(root.get(Author_.id));

// Execute the Query
TypedQuery<Tuple> q = em.createQuery(cq);
List<Tuple> results = q.getResultList();
```

The `size` function is JPA-specific. You can use it to count the elements in a mapped association. As you can see in the log message, Hibernate generates a `JOIN` statement to join the associated table and calls the SQL `count` function to count the number of associated records in the `BookAuthor` table.

```
05:47:23,682 DEBUG [org.hibernate.SQL] -
    select
        author0_.id as col_0_0_,
        count(books1_.authorId) as col_1_0_,
        author0_.id as id1_0_,
        author0_.firstName as firstNam2_0_,
        author0_.lastName as lastName3_0_,
        author0_.version as version4_0_
    from
        Author author0_ cross
    join
        BookAuthor books1_
    where
        author0_.id=books1_.authorId
    group by
        author0_.id
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `CriteriaStandardFunction` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

The Criteria API supports only a subset of the functions supported by the SQL standard and no database-specific functions. Since JPA 2.1, you can use the function `function` to call user-defined or database-specific functions in a `CriteriaQuery` How to call a user-defined function in a CriteriaQuery.

How to call a user-defined function in a CriteriaQuery

Problem

How do I call a user-defined database function in the `WHERE` clause of my CriteriaQuery?

Solution

Since JPA 2.1, you can use the `function(String name, Class<T> type, Expression<?>… args)` method of the `CriteriaBuilder` to call user-defined or database-specific functions. You need to provide the name and the expected result type of the function as the first two parameters, and you can provide one or more `Expression` that will be used as function arguments.

The following code snippet shows an example that calls the user-defined database function `calculate`. The definition of the query follows the same structure as the definition of any other `CriteriaQuery`. I explain it in more detail in How to select entities with a CriteriaQuery.

The definition of the `WHERE` clause is the interesting part of this code sample. I use it to call the user-defined database function `calculate`. Before I can do that, I need to define the two parameters of type `Double` that I want to provide to the database function. The `calculate` function returns a `Double`, and I provide the `price` of the `Book` and a query parameter as function arguments.

```
// Create the CriteriaQuery
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Book> cq = cb.createQuery(Book.class);
Root<Book> root = cq.from(Book.class);
```

```
// call the database function calculate
ParameterExpression<Double> doubleParam1 =
    cb.parameter(Double.class);
ParameterExpression<Double> doubleParam2 =
    cb.parameter(Double.class);
cq.where(
    cb.greaterThan(
        doubleParam2,
        cb.function("calculate", Double.class,
            root.get(Book_.price), doubleParam1)));

// Set bind parameter values and execute the Query
TypedQuery<Book> q = em.createQuery(cq);
q.setParameter(doubleParam1, 10.0D);
q.setParameter(doubleParam2, 40.0D);
List<Book> books = q.getResultList();
```

When you execute this `CriteriaQuery`, Hibernate generates a SQL query with the defined function call.

```
15:08:12,187 DEBUG [org.hibernate.SQL] -
    select
        book0_.id as id1_0_,
        book0_.price as price2_0_,
        book0_.publishingDate as publishi3_0_,
        book0_.title as title4_0_,
        book0_.version as version5_0_
    from
        Book book0_
    where
        ?>calculate(book0_.price, ?)
```

You can use this approach in the `WHERE` clause to call all functions supported by your database.

You can also use the function `function` in the `SELECT` clause of your query. But you then need to register the database function so that Hibernate knows it's result type. That makes the function `function` superfluous because you can use all registered functions directly in your query.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `CriteriaCustomFunction` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

You can also use the function `function` in a JPQL query. I show you how in How to call a user-defined function in a JPQL query.

How to update multiple entities with the Criteria API

Problem

I have to update a lot of entities. With SQL, I would only need 1 `UPDATE` statement to update all entities. Is there a way to do the same with the Criteria API?

Solution

The `CriteriaUpdate` interface allows you to define an update statement that changes multiple entities. The following code snippet shows an example of such a statement. It increases the price of all `Books` by 10%.

```
// define the CriteriaUpdate
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaUpdate<Book> update = cb.createCriteriaUpdate(Book.class);
Root<Book> root = update.from(Book.class);
update.set(Book_.price, cb.prod(root.get(Book_.price), 1.1));

// create and execute the Query
Query query = em.createQuery(update);
query.executeUpdate();
```

As you can see, I define the `CriteriaUpdate` statement in a similar way as a `CriteriaQuery`. I first get a `CriteriaBuilder` from the `EntityManager` and use it to create a `CriteriaUpdate` instance.

Then I define the `Root` of the query in the same way as I do it for a `CriteriaQuery`.

In the next step, I call the `set` method on the `CriteriaUpdate` interface to define the update operation. The first parameter specifies the entity attribute I want to change. Ìn this example, I want to update the `price` attribute. The second parameter is an `Expression` that calculates the new `price`. I want to increase the `price` by 10% and therefore multiply the current `price` with 1.1. That's all you need to do to define a `CriteriaUpdate` statement.

If you don't want to update all entities, you can add a `WHERE` clause to the statement. You can define it in the same way as you do it for a `CriteriaQuery`.

The only step that's left is to create a `Query` based on the `CriteriaUpdate` and execute it. Hibernate then generates a SQL `UPDATE` statement that increases the `price` of all `Books` by 10%. You can check the generated SQL statement in the log file if you activate logging for the SQL statements.

```
18:57:25,118 DEBUG [org.hibernate.SQL] - update Book set
price=price*1.1
```

> Updating multiple entities with a `CriteriaUpdate` statement is more efficient than updating the entities one by one. But Hibernate doesn't know which database records get changed and doesn't update any entities in the first-level cache. You need to make sure that the cache doesn't contain any outdated entities or you need to invalidate the cache programmatically.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `CriteriaUpdate` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

The `CriteriaUpdate` interface is just one option to update multiple entities with one query. You can also use a JPQL update statement as I explain in How to update multiple entities with one JPQL query.

How to delete multiple entities with the Criteria API

Problem

I need to delete multiple entities. Is there a way to do that with the Criteria API that doesn't remove the entities one by one?

Solution

You can use the `CriteriaDelete` interface to remove multiple entities with one SQL statement. The definition of the statement is similar to the definition of a `CriteriaUpdate` or a `CriteriaQuery`. You can see an example in the following code snippet. It deletes all `Book` entities from the database.

```
// define the CriteriaDelete
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaDelete<Book> delete = cb.createCriteriaDelete(Book.class);
delete.from(Book.class);

// create and execute the Query
Query query = em.createQuery(delete);
query.executeUpdate();
```

You first need to get the `CriteriaBuilder` from the `EntityManager` and create a `CriteriaDelete` instance. Then you need to define the `FROM` clause of the delete statement, and you can add an optional `WHERE` clause. You do that in the same way as you do it for a `CriteriaQuery`.

That's all you need to do to define a `CriteriaDelete` statement. You can now use it to create a `Query` and execute it.

Hibernate generates a SQL `DELETE` statement based on your `CriteriaDelete` statement. In this example, it also manages the mapped associations. It first performs an SQL `DELETE` statement to remove all references to the deleted `Book` entities from the association table `BookAuthor`. When that is done, Hibernate executes another SQL statement to remove the `Book` entities.

```
16:49:44,984 DEBUG [org.hibernate.SQL] - delete from BookAuthor
where (bookId) in (select id from Book)
16:49:44,986 DEBUG [org.hibernate.SQL] - delete from Book
```

Deleting multiple entities with a `CriteriaDelete` statement is more efficient than removing them one by one. But Hibernate doesn't know which database records the statement deletes and doesn't remove any entities from the first-level cache. You need to make sure that the cache doesn't contain any deleted entities or you need to invalidate the cache programmatically.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `CriteriaDelete` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

The `CriteriaDelete` interface is just one option to delete multiple entities with one query. You can also use a JPQL delete statement as I explain in How to delete multiple entities with one JPQL query.

How to use pagination with a CriteriaQuery

Problem

The Criteria API does not provide a method to define the number of returned records. How do I paginate the result of a CriteriaQuery?

Solution

Defining pagination for a `CriteriaQuery` is easy but not intuitive, if you're familiar with SQL. The Criteria API doesn't provide any method to define pagination. You have to set the pagination information on the `Query` and not on t h e `CriteriaQuery` interface. You can do that by calling the `setFirstResult(int startPosition)` and `setMaxResults(int maxResults)` methods. This provides the advantage that you can also use this approach to paginate the result of JPQL queries.

The following code snippet shows a simple example that returns the first five `Author` entities from the database. The result set index is 0 based and you need to provide 0 as a `startPosition` to begin with the first element.

```
// Define the CriteriaQuery
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Book> cq = cb.createQuery(Book.class);
Root<Book> root = cq.from(Book.class);
cq.orderBy(cb.asc(root.get(Book_.id)));

// Execute query with pagination
List<Book> books = em.createQuery(cq)
                     .setMaxResults(5)
                     .setFirstResult(0)
                     .getResultList();
```

To select the next five `Authors` from the database, you only need to change the `startPosition` to 5.

```
// Define the CriteriaQuery
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Book> cq = cb.createQuery(Book.class);
Root<Book> root = cq.from(Book.class);
cq.orderBy(cb.asc(root.get(Book_.id)));

// Execute query with pagination
List<Book> books = em.createQuery(cq)
                     .setMaxResults(5)
                     .setFirstResult(5)
                     .getResultList();
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `CriteriaQueryPagination` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn more

You can use the same approach to paginate the result of a JPQL query. I explain it in more detail in How to use pagination with JPQL.

How to reference entity attributes in a type-safe way

Problem

Referencing entity attributes by their name is error-prone and makes the code hard to refactor. How do I reference an attribute in a type-safe way so that I don’t run into problems when I have to change its name?

Solution

I prefer to use the JPA Metamodel to reference entity attributes in a type-safe way. The metamodel consists of a generated class for each managed class in the persistence unit. These classes are stored in the same package and have the same name as the corresponding managed classes with an added "_" at the end.

The only thing you have to do to generate these classes is to add a dependency to Hibernate’s Static Metamodel Generator to your build process like I do in the following Maven `pom.xml` file.

```
<project>
    <modelVersion>4.0.0</modelVersion>

    <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-jpamodelgen</artifactId>
        </dependency>

        ...
    </dependencies>

    ...
</project>
```

Hibernate generates the Metamodel classes with each Maven build and stores them in the `target/generated-sources/annotations` folder of your project.

Let's have a look at an example of a Metamodel class. The first code snippet shows the `Book` entity that you know from most examples in this book. The second one shows the generated metamodel class `Book_`.

```
@Entity
public class Book {

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

    @Version
    private int version;

    private String title;

    private Double price;

    @Temporal(TemporalType.DATE)
    private Date publishingDate;

    @ManyToOne
    @JoinColumn(name="publisherid")
    private Publisher publisher;

    @ManyToMany
    @JoinTable(name="BookAuthor",
        joinColumns={
            @JoinColumn(name="bookId", referencedColumnName="id")},
        inverseJoinColumns={
            @JoinColumn(name="authorId", referencedColumnName="id")
        })
    private Set<Author> authors = new HashSet<Author>();
```

```
    ...
}
```

```
@Generated(
    value="org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor")
@StaticMetamodel(Book.class)
public abstract class Book_ {

    public static volatile SingularAttribute<Book, Double> price;
    public static volatile SingularAttribute<Book, Publisher>
publisher;
    public static volatile SingularAttribute<Book, Long> id;
    public static volatile SingularAttribute<Book, String> title;
    public static volatile SingularAttribute<Book, Date>
publishingDate;
    public static volatile SingularAttribute<Book, Integer> version;
    public static volatile SetAttribute<Book, Author> authors;

}
```

As you can see, the generated class `Book_` has a static attribute for each attribute of the `Book` entity. You can use it to reference the entity attribute in a type-safe way. I do that in the following code snippet to reference the `title` and `publishingDate` attribute of the `Book` entity in a `CriteriaQuery`.

```
// Define CriteriaQuery
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Tuple> cq = cb.createTupleQuery();
Root<Book> root = cq.from(Book.class);

// Use metamodel to reference attributes
cq.multiselect(root.get(Book_.title),
               root.get(Book_.publishingDate));

// Execute Query
List<Tuple> results = em.createQuery(cq).getResultList();
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `JPAMetamodel` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

The folder of the metamodel classes is not a source folder in most IDEs or a standard maven project. You can use a maven plugin to automatically register it as a source folder, as shown in How to automatically add Metamodel classes to your project.

STORED PROCEDURES

Stored procedures are a common approach to implement logic in a database. But in the past, Hibernate and JPA didn't support them well. That changed with JPA 2.1 and the introduction of the `@NamedStoredProcedureQuery` annotation and the `StoredProcedureQuery` interface. You can now define named queries or create ad-hoc queries at runtime to call stored procedures. I show you how to do that in these Hibernate tips:

- How to create an ad-hoc stored procedure call
- How to call a stored procedure with a named query

How to create an ad-hoc stored procedure call

Problem

My database administrator created a stored procedure that I need to call in my business code. How do I create an ad-hoc stored procedure call with Hibernate?

Solution

Since JPA 2.1, you can use the `createStoredProcedureQuery` method of the `EntityManager` to create an ad-hoc stored procedure call.

If you're using a Hibernate version older than 4.3 that doesn't implement JPA 2.1, you need to use a native SQL query to call the stored procedure.

The following code snippet shows an example of an ad-hoc stored procedure call.

```
// define the stored procedure
StoredProcedureQuery query =
    em.createStoredProcedureQuery("calculate");
query.registerStoredProcedureParameter("x", Double.class,
                                       ParameterMode.IN);
query.registerStoredProcedureParameter("y", Double.class,
                                       ParameterMode.IN);
query.registerStoredProcedureParameter("sum", Double.class,
                                       ParameterMode.OUT);

// set input parameter
query.setParameter("x", 1.23d);
query.setParameter("y", 4d);

// call the stored procedure and get the result
query.execute();
Double sum = (Double) query.getOutputParameterValue("sum");
```

You can split an ad-hoc stored procedure call into three parts. You first need to define the stored procedure call, set the input parameters in the second part and then call the stored procedure and retrieve its return value in the final part.

Let's start with the definition of the stored procedure call. You need to call the `createStoredProcedureQuery` method of the `EntityManager` with the name of the stored procedure you want to execute. This method returns a `StoredProcedureQuery` interface that you can use to define a stored procedure call, set its input parameters and call it.

In this example, I call the stored procedure `calculate`. It's a simple procedure that expects the input parameters `x` and `y` and returns the `sum` of them. I call the `registerStoredProcedureParameter` method of the `StoredProcedureQuery` for each of them and register them with their name, type and `ParameterMode`. The `ParameterMode` specifies if the parameter is used as an input (`ParameterMode.IN`), output (`ParameterMode.OUT`), input and output (`ParameterMode.INOUT`) or as a result set cursor (`ParameterMode.REF_CURSOR`).

That's all you need to do to define the stored procedure call. The second and third part of an ad-hoc stored procedure call is identical to the execution of a `@NamedStoredProcedureQuery`.

I set the values for both input parameters by calling the `setParameter` method on the `StoredProcedureQuery` interface for each of them. And then I call the `execute` and the `getOutputParameterValue` methods to execute the stored procedure and to read the return value.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `StoredProcedureQuery` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

If you have to call the stored procedure multiple times in your business logic and don't need to adapt it based on user input, it might be better to use a `@NamedStoredProcedureQuery`. Similar to a named JPQL query, it allows you to define the stored procedure call via annotations and instantiate it by its name. I explain it in more detail in How to call a stored procedure with a named query.

How to call a stored procedure with a named query

Problem

My database administrator created a stored procedure that I need to call from multiple places in my Java code. What's the best way to do that with Hibernate?

Solution

Before JPA 2.0, you had to use a native SQL query to call a stored procedure. In version 2.1, JPA introduced an API and the `@NamedStoredProcedureQuery` annotation to define stored procedure calls. The annotation-based approach is the better solution if you have to call the stored procedure from multiple parts of your business logic without adapting it based on input parameters.

The following code snippet shows an example of a `@NamedStoredProcedureQuery` definition.

```
@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")
    }
)
```

This query calls the stored procedure `calculate` with the input parameters `x` and `y` and the output parameter `sum`. As you can see, each parameter is defined by a `@StoredProcedureParameter` annotation that defines the parameter `mode` and its `name`. The parameter mode specifies if the parameter is used as an input (`ParameterMode.IN`), output (`ParameterMode.OUT`), input and output (`ParameterMode.INOUT`) or as a result set cursor (`ParameterMode.REF_CURSOR`).

That's all you need to do to define the stored procedure call. You can now use it in your business code. You just have to provide its name to the `createNamedStoredProcedureQuery` method of the `EntityManager` to instantiate the query, set the input parameters, execute it, and read the output parameter.

```
// Instantiated NamedStoredProcedureQuery
StoredProcedureQuery query =
    this.em.createNamedStoredProcedureQuery("calculate");

// Set bind parameter values and execute Query
query.setParameter("x", 1.23d);
query.setParameter("y", 4.56d);
query.execute();

// Read return value
Double sum = (Double) query.getOutputParameterValue("sum");
```

Source Code

You can find a project with executable test cases for this Hibernate tip in the `StoredProcedureQuery` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

A `@NamedStoredProcedureQuery` is easy to use, but you can't adapt it at runtime. If you need more flexibility, you should have a look at the `StoredProcedureQuery` API. You can use it to define a stored procedure call at runtime, which requires more code but also provides more flexibility. I explain the it in How to create an ad-hoc stored procedure call.

CACHING

The general idea of caching is pretty simple: You store often used entities or query results in the memory of your business tier instead of the database. If done correctly, this can avoid a lot of database requests and improve the performance of your application. The Hibernate tips in this chapter show you how to activate and use Hibernate's second-level and query cache:

- How to store an entity in the second-level cache
- How to use the query cache to avoid additional queries

If you like to dive deeper into these topics, please take a look at my Hibernate Performance Tuning Online Training. It gets into a lot more details about the general concept of Hibernate's different caches and their pitfalls.

How to store an entity in the second-level cache

Problem

Several of my use cases read the same entities from the database without changing them. How do I cache them in my application so that Hibernate doesn't always have to get them from the database?

Solution

You can use Hibernate's session-independent second-level cache to cache entities that are used by multiple Hibernate `Session`. The cache is deactivated by default, and you need to set two configuration parameters to activate it:

1. You need to define the `shared-cache-mode`. It tells Hibernate how to select entities for the second-level cache. You can choose between `ALL` to cache all entities, `NONE` to don't cache any entities, `ENABLE_SELECTIVE` to select the entities that will be cached and `DISABLE_SELECTIVE` to select the entities that will not be cached. I prefer `ENABLE_SELECTIVE` because it requires me to explicitly decide which entities are cached.
2. You need to specify the caching provider you want to use by setting the `hibernate.cache.region.factory_class` property.

The following code snippet shows parts of a `persistence.xml` file with a second-level cache configuration. It uses the `shared_cache_mode` `ENABLE_SELECTIVE` and EhCache as the caching provider.

```
<persistence>
    <persistence-unit name="my-persistence-unit">
        ...

        <!-- enable selective 2nd level cache -->
        <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>

        <properties>
            ...

            <!-- configure caching -->
            <property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.ehcache.EhCacheRegionFactory"/>
        </properties>
    </persistence-unit>
</persistence>
```

If you use `ENABLE_SELECTIVE` as your `shared-cache-mode` you need to annotate the entities you want to cache with JPA's `@Cacheable` or Hibernate's `@Cache` annotation.

```
@Entity
@Cacheable
public class Author {
    ...
}
```

You can also use the `@Cacheable(false)` annotation to exclude an entity from caching if you use `shared_cache_mode` `DISABLE_SELECTIVE`.

That's all you need to do to activate the second-level cache for your application. Hibernate now stores all cacheable entities you've used in the second-level cache. When your next use case calls the `find` method on the `EntityManager` or when Hibernate needs to retrieve an entity to initialize an

entity association, it tries to get it from the second-level cache before it performs a database query.

> Hibernate doesn't use the second-level cache with a JPQL or Criteria query.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `2ndLevelCache` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

The second-level cache stores entities and no query results. If you want to cache the result of a query, you can use Hibernate's Query Cache. I show you how in How to use the query cache to avoid additional queries.

How to use the query cache to avoid additional queries

Problem

Hibernate does not use the first- and second-level cache for queries. Is there a way to cache the result of a query?

Solution

Hibernate also supports the query cache, which stores query results.

> The result of a query depends on the executed SQL statement and the used parameter values. The cache, therefore, stores the result for a query with a specific set of parameters. Make sure that you often use the same parameter values before you decide to cache the result of a query.

If you identify a query that you often execute with the same parameter values, you need to activate the query cache in the `persistence.xml` file. You do that by setting the parameter `hibernate.cache.use_query_cache` to true and defining a `hibernate.cache.region.factory_class`.

```
<persistence>
        <persistence-unit name="my-persistence-unit">
                ...
                <properties>
                        ...
                        <!--  configure caching -->
                        <property
name="hibernate.cache.use_query_cache"
value="true"/>
                        <property
name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.ehcache.EhCacheRegionFactory"/>
```

```
                </properties>
        </persistence-unit>
</persistence>
```

You also need to activate caching for the query whose results you want to cache. You can do that by calling the `setCacheable` method of the Hibernate-specific Query interface using the parameter `true`.

```
Session s = (Session) em.getDelegate();
Query q = s.createQuery("SELECT a FROM Author a WHERE id = :id");
q.setParameter("id", 1L);
q.setCacheable(true);
Author a = q.uniqueResult();
```

Hibernate now stores the result of this query in the query cache. When your next use case executes this query, Hibernate checks if the query cache contains a result for the given parameter values. If the cache contains a matching record, Hibernate gets the query result from there. Otherwise, it executes the query and store its result in the cache.

Source Code

You can find a project with executable test cases for this Hibernate tip in the `QueryCache` module of the example project. If you haven't already done so, you can download it at http://www.hibernate-tips.com/download-examples.

Learn More

The query cache stores query results and no entities. If you want to cache entities that you often use in your business code, you can use the second-level cache. I show you how in How to store an entity in the second-level cache.

ACKNOWLEDGMENTS

I would like to thank all reviewers, and especially Frits Walraven. Writing a book is just the first step. It takes a lot of time, effort, and technical knowledge to get from the first version to the final book. Thank you for your support and all the feedback you provided.

Thank you to Steve Ebersole, who immediately agreed to write the foreword and provided great feedback on the book.

I would also like to thank Michael Simons for helping me with the How to bootstrap Hibernate with Spring Boot tip.

Thank you to all the readers of www.thoughts-on-java.org. Your support convinced me to quit my day job and follow my passion for writing and teaching. Your interest and feedback on the Hibernate Tips series and this book kept me going and reasssured me that I was doing something meaningful.

I also want to thank Nermina Miller, who edited this book and fixed all my spelling, grammar, and punctuation mistakes.

Thank you to Dan Allen, who helped me with Asciidoctor. Without your incredible support, I wouldn’t have been able to finish the book in time.

Thank you to Tom Oberbichler, who guided me through the creation process of this book and helped me to focus on the right tasks at the right time. Your support gave me the peace of mind that I had the most important things under control even when it felt like I had to work on all of them at once.

And, a special thank you goes to my amazing wife Sandra, who encouraged me in so many ways over the years. She’s always there and supports me in any way she can, even as I spend nights and weekends writing books or recording online courses. Without you, I would have never been able to do it all.
Thanks also to my little son, Lars, who reminds me that there are so many other fantastic things in life.
I love you both!

Thorben

REVIEWERS

Special thanks to my technical reviewers who helped me immensely with their feedback.

Frits Walraven

Frits has been a software engineer for about 15 years. He got his MSc in Computer Science from the Technical University in Twente, Netherlands, back in 1996. After a few years of coding in C, he jumped into the big Java world. Nowadays he is mostly involved with functional design but tries to keep his coding alive through Oracle Certification. He currently holds the SCJP5, SCWCD5, OCEEJB6, and OCEWSD6 certificates and enjoys writing and sharing his notes. He is the author of the Enthuware Java Web Services Developer EE6 mock exams.

Mark Spritzler

Mark Spritzler is a former JBoss and SpringSource staff member, and has been teaching enterprise software for more than 25 years as a software developer. He has spent the last 13 years writing Hibernate/JPA applications on many enterprise applications.

Petri Kainulainen

Petri Kainulainen is passionate about software development and continuous improvement. He specialized in software development using the Spring Framework and is the author of the Test With Spring online course.

Sandra Janssen

Sandra Janssen has studied technomathematics and worked several years at a university with a strong focus on scientific computing. She is a regular reviewer of content published on Thoughts on Java. Together with Thorben Janssen, she works on the strategic alignment of the Thoughts on Java platform and all related activities.

THORBEN JANSSEN

Thorben Janssen has been a software developer and architect for more than 15 years. More than a decade ago, he used one of the first Hibernate releases to implement the persistence layer of enterprise applications. Since then, he has used the framework to implement applications of various sizes with complex business and performance requirements.

After blogging for several years about JPA and Hibernate, he decided to quit his day job in October 2016 to follow his passion for writing and teaching. Since then, he has been working as an independent trainer, author, and consultant to show software developers how to use Hibernate and JPA to avoid common problems and implement their persistence layer with ease.