Mục lục

[Hibernate Framework là gì? 4](#_Toc35601462)

[Lợi ích của Hibernate Framework 4](#_Toc35601463)

[Kiến trúc Hibernate 5](#_Toc35601464)

[Các thành phần trong kiến trúc Hibernate 6](#_Toc35601465)

[JDBC là gì? 8](#_Toc35601466)

[Ưu và nhược điểm của JDBC 8](#_Toc35601467)

[Tại sao cần lập bản đồ quan hệ đối tượng (ORM)? 8](#_Toc35601468)

[ORM là gì? 10](#_Toc35601469)

[Java ORM Frameworks 11](#_Toc35601470)

[Tạo một Chương trình Hibernate với InteliJ IDEA 11](#_Toc35601471)

[Cấu Hình Hibernate - Configuration 13](#_Toc35601472)

[Các thuộc tính Hibernate 13](#_Toc35601473)

[Ví dụ về Hibernate kết hợp với MySQL 15](#_Toc35601474)

[Session 19](#_Toc35601475)

[Định nghĩa và sử dụng 19](#_Toc35601476)

[Các phương thức của interface Session 20](#_Toc35601477)

[Hibernate - Persistent Class 22](#_Toc35601478)

[Một ví dụ về Java POJO đơn giản 23](#_Toc35601479)

[Hibernate - Mapping Files 24](#_Toc35601480)

[Ví dụ file mapping XML trong Hiberante 24](#_Toc35601481)

[Các thẻ trong file mapping 25](#_Toc35601482)

[Hibernate - Mapping Types 26](#_Toc35601483)

[Các kiểu dữ liệu nguyên thủy 26](#_Toc35601484)

[Các kiểu Date and time 27](#_Toc35601485)

[Các kiểu Binary và đối tượng lớn 27](#_Toc35601486)

[Các kiểu JDK-related 28](#_Toc35601487)

[Hibernate – Examples 29](#_Toc35601488)

[Hibernate - O/R Mappings 29](#_Toc35601489)

[Collections Mappings 29](#_Toc35601490)

[Association Mappings 30](#_Toc35601491)

[Component Mappings 30](#_Toc35601492)

[Hibernate - Annotations 30](#_Toc35601493)

[Tạo lớp Entity với các annotation 31](#_Toc35601494)

[@Entity Annotation: 31](#_Toc35601495)

[@Table Annotation: 32](#_Toc35601496)

[@Id and @GeneratedValue Annotations: 32](#_Toc35601497)

[@Column Annotation: 32](#_Toc35601498)

[Hibernate - Query Language 32](#_Toc35601499)

[Mệnh đề FROM 33](#_Toc35601500)

[Mệnh đề AS 33](#_Toc35601501)

[Mệnh đề SELECT 33](#_Toc35601502)

[Mệnh đề WHERE 34](#_Toc35601503)

[Mệnh đề ORDER BY 34](#_Toc35601504)

[Mệnh đề GROUP BY 34](#_Toc35601505)

[Sử dụng các tham số đặt tên 34](#_Toc35601506)

[Mệnh đề UPDATE 35](#_Toc35601507)

[Mệnh đề DELETE 35](#_Toc35601508)

[Mệnh đề INSERT 35](#_Toc35601509)

[Các phương thức tổng hợp 36](#_Toc35601510)

[Phân trang bằng việc sử dụng Query 36](#_Toc35601511)

[Hibernate - Criteria Queries 37](#_Toc35601512)

[Restrictions với Criteria 37](#_Toc35601513)

[Phân trang bằng việc sử dụng Criteria 38](#_Toc35601514)

[Sắp xếp kết quả 39](#_Toc35601515)

[Projections & Aggregations 39](#_Toc35601516)

[Ví dụ về câu truy vấn Criteria 40](#_Toc35601517)

[Hibernate - Native SQL 43](#_Toc35601518)

[Các kiểu truy vấn Native SQL 44](#_Toc35601519)

[1. Truy vấn vô hướng (Scalar) 44](#_Toc35601520)

[2. Truy vấn Entity 44](#_Toc35601521)

[Ví dụ về Native SQL trong Hibernate 44](#_Toc35601522)

[Hibernate - Caching 45](#_Toc35601523)

[Các kiểu bộ nhớ cache trong Hibernate 46](#_Toc35601524)

[Chiến lược truy cập đồng thời 47](#_Toc35601525)

[Cache provider 48](#_Toc35601526)

[Bộ nhớ cache truy vấn 50](#_Toc35601527)

[Hibernate - Batch Processing 51](#_Toc35601528)

[Ví dụ về Batch Processing trong Hibernate 52](#_Toc35601529)

[Hibernate - Interceptors 53](#_Toc35601530)

[Ví dụ về Interceptor trong Hibernate 54](#_Toc35601531)

[Tạo các Interceptor 54](#_Toc35601532)

[Tạo lớp POJO 55](#_Toc35601533)

[Tạo bảng cơ sở dữ liệu 56](#_Toc35601534)

[Tạo file mapping 56](#_Toc35601535)

## Hibernate Framework là gì?

Hibernate framework là một giải pháp ORM (Object Relational Mapping) mã nguồn mở, gọn nhẹ. Hibernate giúp đơn giản hoá sự phát triển của ứng dụng java để tương tác với cơ sở dữ liệu.

Tool ORM giúp đơn giản hoá việc tạo ra dữ liệu, thao tác dữ liệu và truy cập dữ liệu. Đó là một kỹ thuật lập trình để ánh xạ đối tượng vào dữ liệu được lưu trữ trong cơ sở dữ liệu.



Các tool ORM sử dụng JDBC API để tương tác với cơ sở dữ liệu.

### Lợi ích của Hibernate Framework

Hibernate Framework có các lợi ích như dưới đây:

**1. Mã nguồn mở và nhẹ:**Hibernate Framework là mã nguồn mở có giấy phép LGPL và nhẹ.

**2. Hiệu suất nhanh:** Hiệu suất của Hibernate Framework là nhanh bởi vì bộ nhớ cache được sử dụng trong nội bộ Hibernate Framework. Có hai loại bộ nhớ cache trong Hibernate Framework, gồm bộ nhớ cache cấp một và bộ nhớ cache cấp hai. Bộ nhớ cache cấp một được bật bằng lệnh mặc định.

**3. Truy vấn cơ sở dữ liệu độc lập:** HQL (Hibernate Query Language) là phiên bản hướng đối tượng của SQL. Nó tạo ra các truy vấn cơ sở dữ liệu độc lập. Vì vậy, bạn không cần phải viết các truy vấn cơ sở dữ liệu cụ thể. Trước Hibernate, nếu dự án có cơ sở dữ liệu bị thay đổi, chúng ta cần phải thay đổi truy vấn SQL dẫn đến sự cố bảo trì.

**4. Tạo bảng tự động:** Hibernate framework cung cấp phương tiện để tạo ra các bảng cơ sở dữ liệu tự động. Vì vậy, không cần phải tạo ra các bảng trong cơ sở dữ liệu bằng tay.

**5. Đơn giản lệnh join phức tạp:** Có thể lấy dữ liệu từ nhiều bảng một cách dễ dàng với Hibernate framework.

**6. Cung cấp thống kê truy vấn và trạng thái cơ sở dữ liệu:** Hibernate hỗ trợ bộ nhớ cache truy vấn và cung cấp số liệu thống kê về truy vấn và trạng thái cơ sở dữ liệu.

## Kiến trúc Hibernate

Kiến trúc Hibernate bao gồm nhiều đối tượng như đối tượng persistent, session factory, transaction factory, connection factory, session, transaction,...

Có 4 tầng trong kiến trúc hibernate đó là tầng ứng dụng java, tầng hibernate framework, tầng backhand api và tầng database. Dưới đây là sơ đồ kiến trúc hibernate:



Dưới đây là một hình ảnh chi tiết về Kiến trúc ứng dụng Hibernate với vài lớp core quan trọng.



Hibernate sử dụng các API Java hiện có, như JDBC, Java Transaction API (JTA) và Java Naming and Directory Interface (JNDI). JDBC cung cấp một mức độ trừu tượng cơ bản của cơ sở dữ liệu quan hệ, cho phép hầu hết các cơ sở dữ liệu với trình điều khiển JDBC được hỗ trợ bởi Hibernate. JNDI và JTA cho phép Hibernate được tích hợp với các máy chủ ứng dụng J2EE.

### Các thành phần trong kiến trúc Hibernate

Để tạo ứng dụng hibernate đầu tiên, chúng ta phải biết các thành phần của kiến trúc Hibernate. Chúng là như sau:

#### Đối tượng Configuration

Đối tượng Configuration là đối tượng Hibernate đầu tiên bạn tạo trong bất kỳ ứng dụng Hibernate nào và chỉ cần tạo một lần trong quá trình khởi tạo ứng dụng. Nó đại diện cho một tập tin cấu hình hoặc thuộc tính yêu cầu của Hibernate. Đối tượng Configuration cung cấp hai thành phần chính:

* **Database Connection:** Thao tác này được xử lý thông qua một hoặc nhiều tệp cấu hình được Hibernate hỗ trợ. Các tệp này là hibernate.properties và hibernate.cfg.xml.
* **Class Mapping Setup:** Thành phần này tạo ra kết nối giữa các lớp Java và các bảng cơ sở dữ liệu.

#### Đối tượng SessionFactory

Đối tượng Configuration được sử dụng để tạo ra một đối tượng SessionFactory mà cấu hình Hibernate cho ứng dụng bằng cách sử dụng tệp tin cấu hình được cung cấp và cho phép một đối tượng Session được khởi tạo. SessionFactory là một đối tượng luồng an toàn và được sử dụng bởi tất cả các luồng của một ứng dụng.

SessionFactory là đối tượng nặng nên thường nó được tạo ra trong quá trình khởi động ứng dụng và lưu giữ để sử dụng sau này. Bạn sẽ cần một đối tượng SessionFactory cho mỗi cơ sở dữ liệu bằng cách sử dụng một tập tin cấu hình riêng biệt. Vì vậy, nếu bạn đang sử dụng nhiều cơ sở dữ liệu thì bạn sẽ phải tạo nhiều đối tượng SessionFactory.

#### Đối tượng Session

Một session được sử dụng để có được một kết nối vật lý với một cơ sở dữ liệu. Đối tượng Session là nhẹ và được thiết kế để được tạo ra thể hiện mỗi khi tương tác với cơ sở dữ liệu. Các đối tượng liên tục được lưu và truy xuất thông qua một đối tượng Session.

Các đối tượng Session không nên được mở trong một thời gian dài bởi vì chúng thường không phải là luồng an toàn và chúng cần được tạo ra và được đóng khi cần thiết.

#### Đối tượng Transaction

Một Transaction đại diện cho một đơn vị làm việc với cơ sở dữ liệu và hầu hết các RDBMS hỗ trợ chức năng transaction. Các transaction trong Hibernate được xử lý bởi trình quản lý transaction và transaction (từ JDBC hoặc JTA).

Đây là một đối tượng tùy chọn và các ứng dụng Hibernate có thể chọn không sử dụng interface này, thay vào đó quản lý transaction trong code ứng dụng riêng.

#### Đối tượng Query

Các đối tượng Query sử dụng chuỗi truy vấn SQL hoặc Hibernate Query Language (HQL) để lấy dữ liệu từ cơ sở dữ liệu và tạo các đối tượng. Ví dụ truy vấn được sử dụng để ràng buộc các tham số truy vấn, giới hạn số lượng kết quả được trả về bởi truy vấn và cuối cùng thực hiện truy vấn.

#### Đối tượng Criteria

Đối tượng Criteria được sử dụng để tạo và thực hiện truy vấn các tiêu chí định hướng đối tượng để lấy các đối tượng.

## JDBC là gì?

JDBC là viết tắt của **Java Database Connectivity**. Nó cung cấp một bộ API Java để truy cập cơ sở dữ liệu quan hệ từ chương trình Java. Các API Java này cho phép các chương trình Java thực thi các câu lệnh SQL và tương tác với bất kỳ cơ sở dữ liệu tuân thủ SQL nào.

JDBC cung cấp một kiến ​​trúc linh hoạt để viết một ứng dụng độc lập với cơ sở dữ liệu có thể chạy trên các nền tảng khác nhau và tương tác với các DBMS khác nhau mà không cần sửa đổi.

### Ưu và nhược điểm của JDBC

|  |  |
| --- | --- |
| **Ưu điểm của JDBC** | **Nhược điểm của JDBC** |
| Xử lý SQL đơn giản và gọn gàng  Hiệu suất tốt với dữ liệu lớn  Rất tốt cho các ứng dụng nhỏ  Cú pháp đơn giản để dễ học | Phức tạp nếu nó được sử dụng trong các dự án lớn  Chi phí lập trình lớn  Không đóng gói  Khó thực hiện khái niệm MVC  Truy vấn là DBMS cụ thể |

### Tại sao cần lập bản đồ quan hệ đối tượng (ORM)?

Khi chúng ta làm việc với một hệ thống hướng đối tượng, giữa mô hình đối tượng và cơ sở dữ liệu quan hệ sẽ có một sự không phù hợp. Các RDBMS biểu thị dữ liệu theo định dạng bảng trong khi các ngôn ngữ hướng đối tượng, chẳng hạn như Java hay C # biểu thị nó dưới dạng biểu đồ liên kết của các đối tượng.

Hãy xem xét Lớp Java sau với các hàm tạo thích hợp và hàm công khai được liên kết

*public class* Employee {  
 *private int* id;  
 *private* String first\_name;  
 *private* String last\_name;  
 *private int* salary;  
}

Các đối tượng trên sẽ được lưu trữ và truy xuất vào bảng RDBMS sau

*create table* EMPLOYEE (  
 id *INT NOT NULL auto\_increment*,  
 first\_name *VARCHAR*(20) *default NULL*,  
 last\_name *VARCHAR*(20) *default NULL*,  
 salary *INT default NULL*,  
 *PRIMARY KEY* (id)  
);

Đầu tiên, sẽ ra sao nếu chúng ta cần phải sửa đổi thiết kế cơ sở dữ sau khi đã phát triển vài trang hoặc ứng dụng? Thứ hai, việc tải và lưu trữ các đối tượng trong một cơ sở dữ liệu quan hệ làm cho chúng ta gặp 5 vấn đề không khớp (mismatch).

|  |  |
| --- | --- |
| **No.** | **Mismatch & Mô tả** |
| 1 | **Tính chi tiết**  Đôi khi bạn sẽ có một mô hình đối tượng có nhiều lớp hơn số lượng các bảng tương ứng trong cơ sở dữ liệu. |
| 2 | **Tính kế thừa**  Các RDBMS không định nghĩa bất cứ điều gì tương tự như Kế thừa, đó là một mô hình tự nhiên trong các ngôn ngữ lập trình hướng đối tượng. |
| 3 | **Identity**  Một RDBMS định nghĩa chính xác một khái niệm về 'sự giống nhau': khóa chính. Tuy nhiên, Java định nghĩa cả danh tính đối tượng (a == b) và đẳng thức đối tượng (a.equals (b)). |
| 4 | **Mối quan hệ kết hợp**  Các ngôn ngữ hướng đối tượng biểu diễn cho các sự kết hợp sử dụng các đối tượng tham chiếu, trong khi RDBMS biểu diễn mối quan hệ kết hợp bằng một cột khoá ngoại. |
| 5 | **Điều hướng**  Các cách bạn truy cập các đối tượng trong Java và trong một RDBMS về cơ bản là khác nhau. |

Object- Relatic Mapping (ORM) là giải pháp để xử lý tất cả các vấn đề ở trên.

## ORM là gì?

ORM là viết tắt của Object-Relational Mapping (ORM) là một kỹ thuật lập trình để chuyển đổi dữ liệu giữa các cơ sở dữ liệu quan hệ và các ngôn ngữ lập trình hướng đối tượng như Java, C# ... Một hệ thống ORM có những ưu điểm so vơi JDBC như sau:

|  |  |
| --- | --- |
| **No.** | **Ưu điểm** |
| 1 | Cho phép đối tượng truy cập mã nghiệp vụ thay vì bảng DB. |
| 2 | Ẩn các chi tiết của các truy vấn SQL từ logic OO. |
| 3 | Dựa trên JDBC 'under the hood' |
| 4 | Không cần phải đối phó với việc thực hiện cơ sở dữ liệu. |
| 5 | Các thực thể dựa trên các khái niệm nghiệp vụ thay vì cấu trúc cơ sở dữ liệu. |
| 6 | Quản lý Transaction và tạo ra key tự động. |
| 7 | Phát triển ứng dụng nhanh chóng. |

Một giải pháp ORM bao gồm bốn thực thể sau:

|  |  |
| --- | --- |
| **No.** | **Giải pháp** |
| 1 | Một API để thực hiện các thao tác CRUD cơ bản trên các đối tượng của các lớp persistent. |
| 2 | Một ngôn ngữ hoặc API để chỉ định các truy vấn đề cập đến các lớp và thuộc tính của các lớp. |
| 3 | Một phương tiện có thể cấu hình để chỉ định siêu dữ liệu ánh xạ. |
| 4 | Một kỹ thuật tương tác với các đối tượng giao dịch để thực hiện dirty checking, lazy loading, join fetching và các chức năng tối ưu hóa khác. |

### Java ORM Frameworks

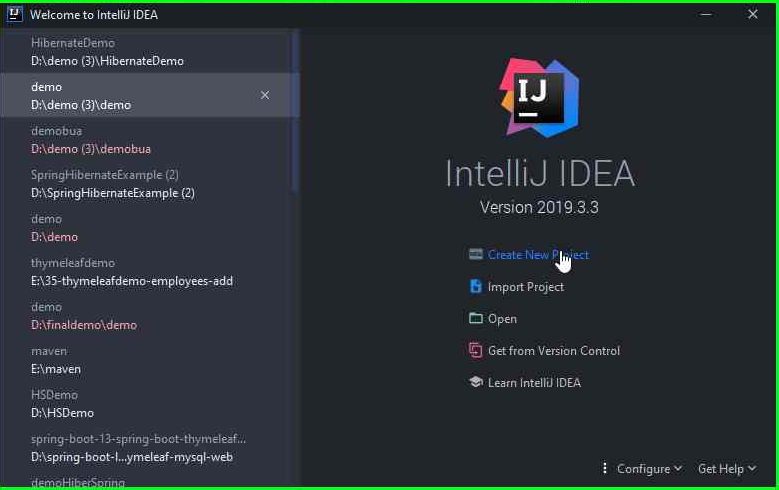
Có rất nhiều persistent frameworks và lựa chọn ORM cho Java. Một persistent framework là một dịch vụ ORM lưu trữ và truy xuất các đối tượng vào cơ sở dữ liệu quan hệ, như là:

* Enterprise JavaBeans Entity Beans
* Java Data Objects
* Castor
* TopLink
* Spring DAO
* Hibernate
* Vân vân…

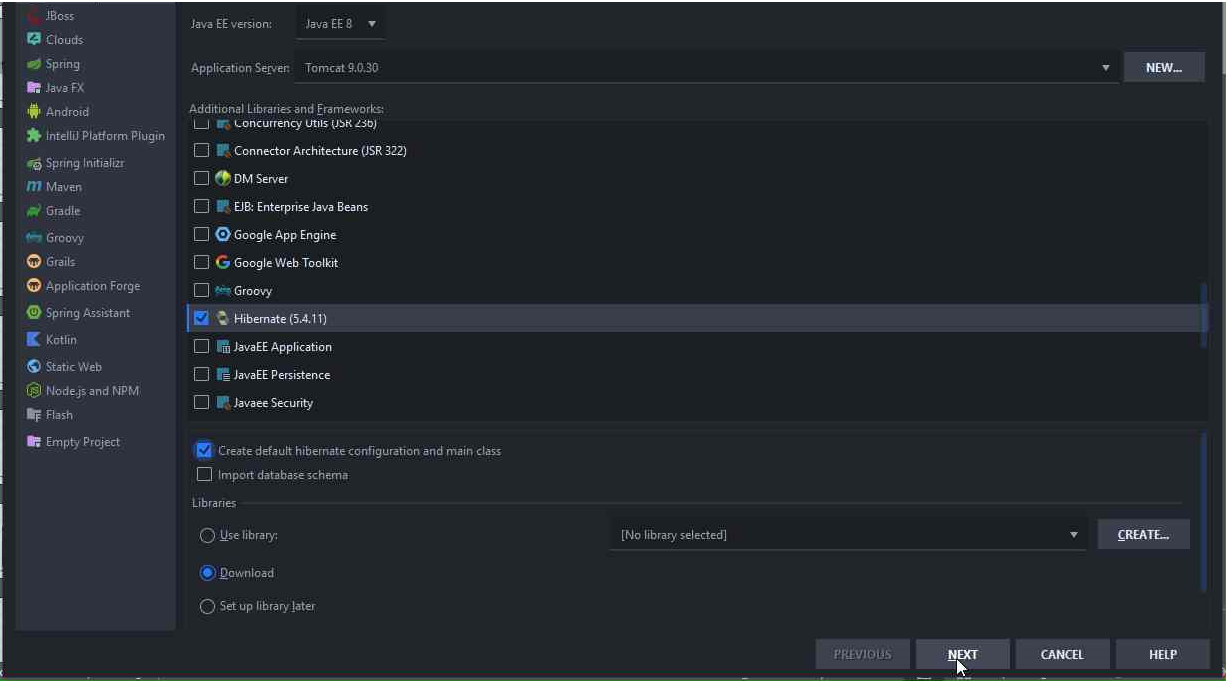
## Tạo một Chương trình Hibernate với InteliJ IDEA

Bạn có thể cài đặt Hibernate theo cách thủ công tại <http://www.hibernate.org/downloads> hoặc sử dụng IDE. Có nhiều IDE có thể sử dụng để cài đặt Hibernate. Tuy nhiên ở đây chúng ta sẽ lấy ví dụ với InteliJ.

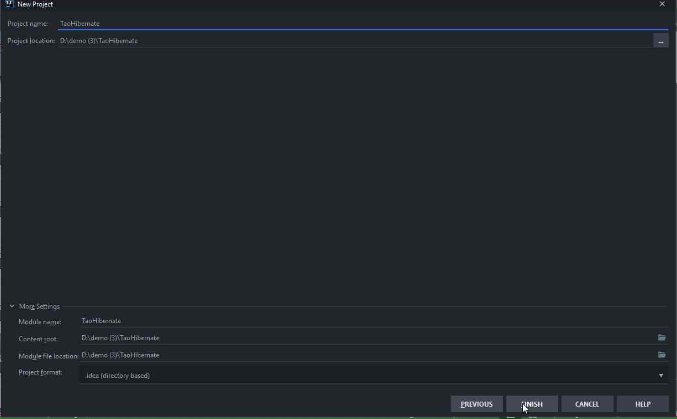
Đầu tiên, ta chọn Create Project



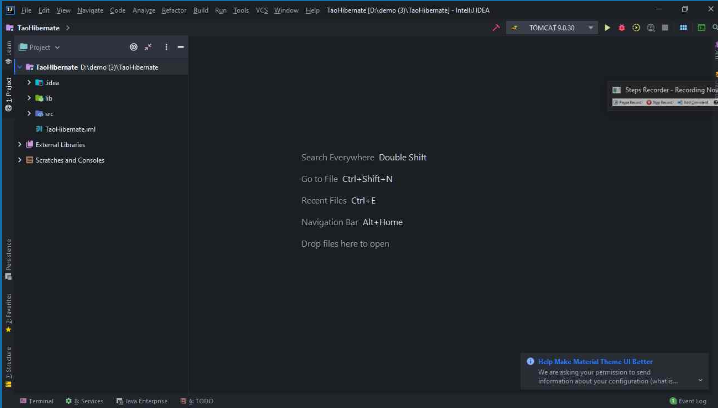
Sau đó tìm tới Java EE, chọn Hibernate, sau đó tick vào create default hibernate configuration and main class.



Sau khi nhấn next, ta đặt tên cho chương trình và finish.



Sau khi dowload những file cần thiết, ta sẽ được chương trình Hibernate đầu tiên.



## Cấu Hình Hibernate - Configuration

Hibernate yêu cầu biết trước các thông tin các để lớp Java ánh xạ đến các bảng cơ sở dữ liệu. Hibernate cũng yêu cầu một tập hợp các thiết lập cấu hình liên quan đến cơ sở dữ liệu và các tham số liên quan khác. Tất cả các thông tin như vậy thường được cung cấp dưới dạng file thuộc tính Java tiêu chuẩn có tên gọi là **hibernate.properties**, hoặc dưới dạng file XML có tên **hibernate.cfg.xml**.

Chúng ta sử dụng file XML **hibernate.cfg.xml** để chỉ định các thuộc tính Hibernate bắt buộc trong các ví dụ trong các bài học Hibernate. Hầu hết các thuộc tính lấy giá trị mặc định của chúng và không bắt buộc phải chỉ định chúng trong file properties trừ khi nó thực sự là bắt buộc. Tập tin này được lưu trong thư mục gốc của ứng dụng.

## Các thuộc tính Hibernate

Dưới đây là danh sách các thuộc tính quan trọng mà bạn sẽ cần để cấu hình cho một cơ sở dữ liệu trong một trường hợp standalone:

|  |  |
| --- | --- |
| **No.** | **Các thuộc tính và mô tả** |
| 1 | **hibernate.dialect** Thuộc tính này làm cho Hibernate tạo ra SQL thích hợp cho cơ sở dữ liệu đã chọn. |
| 2 | **hibernate.connection.driver\_class**  JDBC driver class. |
| 3 | **hibernate.connection.url**  JDBC URL của cơ sở dữ liệu. |
| 4 | **hibernate.connection.username**  username của cơ sở dữ liệu. |
| 5 | **hibernate.connection.password**  password của cơ sở dữ liệu. |
| 6 | **hibernate.connection.pool\_size**  Giới hạn số kết nối đang chờ trong pool kết nối cơ sở dữ liệu Hibernate. |
| 7 | **hibernate.connection.autocommit**  Cho phép chế độ autocommit để được sử dụng cho JDBC connection. |

Nếu bạn đang sử dụng một cơ sở dữ liệu cùng với một máy chủ ứng dụng và JNDI thì bạn phải cấu hình các thuộc tính sau:

|  |  |
| --- | --- |
| **No.** | **Các thuộc tính và mô tả** |
| 1 | **hibernate.connection.datasource**  Tên JNDI được định nghĩa trong ngữ cảnh máy chủ ứng dụng mà bạn đang sử dụng cho ứng dụng. |
| 2 | **hibernate.jndi.class**  Lớp InitialContext cho JNDI. |
| 3 | **hibernate.jndi.<JNDIpropertyname>**  Truyền bất kỳ thuộc tính JNDI bạn thích tới JNDI *InitialContext.* |
| 4 | **hibernate.jndi.url**  Cung cấp URL cho JNDI. |
| 5 | **hibernate.connection.username**  username của cơ sở dữ liệu. |
| 6 | **hibernate.connection.password**  password của cơ sở dữ liệu. |

### Ví dụ về Hibernate kết hợp với MySQL

Sau đây là một ví dụ về file **hibernate.cfg.xml.**  Giúp liên kết Hibernate với cơ sở dữ liệu MySQL.

<?*xml version*='1.0' *encoding*='utf-8'?>  
<!DOCTYPE *hibernate-configuration* PUBLIC  
 "-//Hibernate/Hibernate Configuration DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  
<hibernate-configuration>  
 <session-factory>

<property *name*="connection.url">jdbc:mysql://localhost:3306/demo</property>

<property *name*="connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

<property *name*="dialect">org.hibernate.dialect.MySQLDialect</property>

<property *name*="current\_session\_context\_class">thread</property>

<property *name*="connection.username">taikhoanmysql</property>

<property *name*="connection.password">matkhaumysql</property>

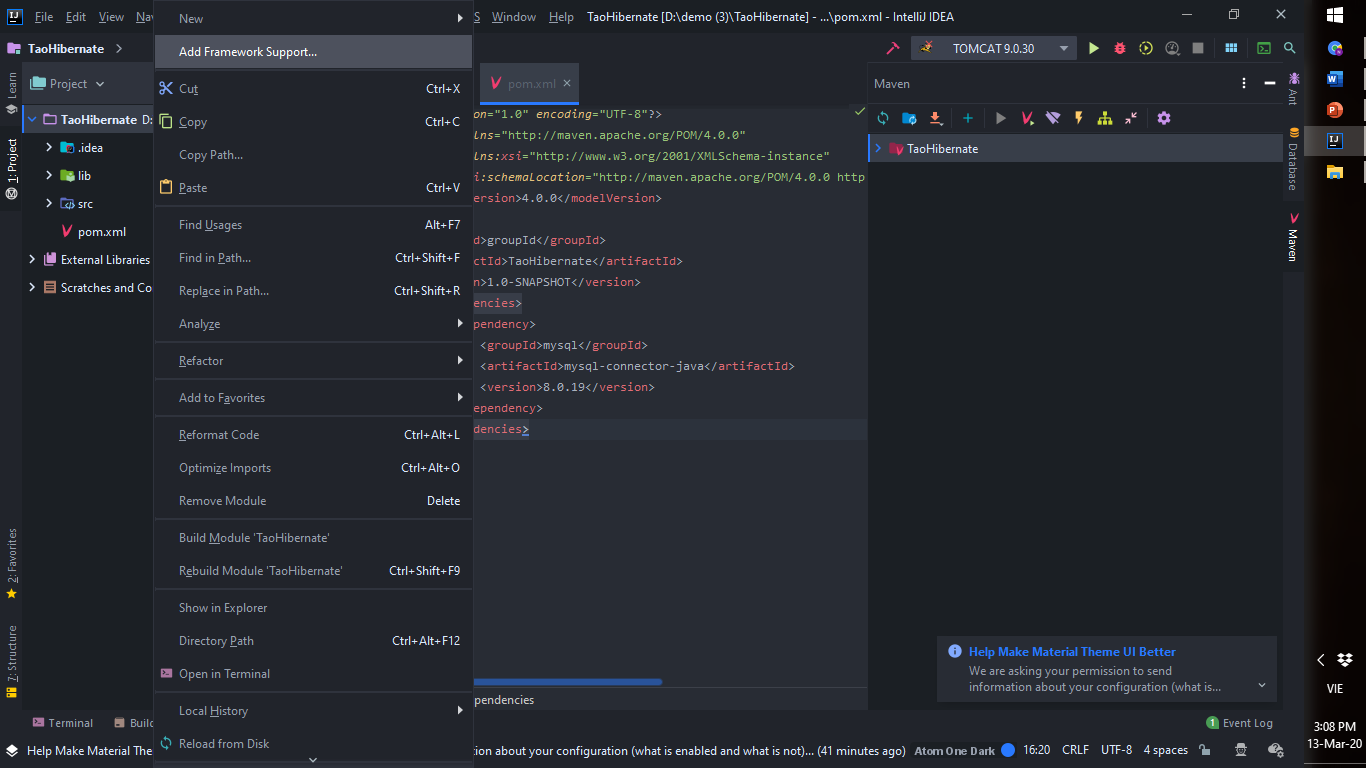
</session-factory>  
  
  
</hibernate-configuration>

Ở phần **dialect**, vì ta sử dụng MySQL nên sử dụng org.hibernate.dialect.MySQLDialect , ngoài ra khi sử dụng các database khác ta có thể thay đổi các giá trị tương ứng với:

|  |  |
| --- | --- |
| **Sr.No.** | **Database & Dialect Property** |
| 1 | **DB2**  org.hibernate.dialect.DB2Dialect |
| 2 | **HSQLDB**  org.hibernate.dialect.HSQLDialect |
| 3 | **HypersonicSQL**  org.hibernate.dialect.HSQLDialect |
| 4 | **Informix**  org.hibernate.dialect.InformixDialect |
| 5 | **Ingres**  org.hibernate.dialect.IngresDialect |
| 6 | **Interbase**  org.hibernate.dialect.InterbaseDialect |
| 7 | **Microsoft SQL Server 2000**  org.hibernate.dialect.SQLServerDialect |
| 8 | **Microsoft SQL Server 2005**  org.hibernate.dialect.SQLServer2005Dialect |
| 9 | **Microsoft SQL Server 2008**  org.hibernate.dialect.SQLServer2008Dialect |
| 10 | **MySQL**  org.hibernate.dialect.MySQLDialect |
| 11 | **Oracle (any version)**  org.hibernate.dialect.OracleDialect |
| 12 | **Oracle 11g**  org.hibernate.dialect.Oracle10gDialect |
| 13 | **Oracle 10g**  org.hibernate.dialect.Oracle10gDialect |
| 14 | **Oracle 9i**  org.hibernate.dialect.Oracle9iDialect |
| 15 | **PostgreSQL**  org.hibernate.dialect.PostgreSQLDialect |
| 16 | **Progress**  org.hibernate.dialect.ProgressDialect |
| 17 | **SAP DB**  org.hibernate.dialect.SAPDBDialect |
| 18 | **Sybase**  org.hibernate.dialect.SybaseDialect |
| 19 | **Sybase Anywhere**  org.hibernate.dialect.SybaseAnywhereDialect |

Chú ý:

Và **conection.driver\_class**, ta cần thêm driver cần thiết vào thì chương trình mới có thể hoạt động. Ta có thể sử dụng Maven, bằng cách chuột phải vào Project, -> add Framework Support, sau đó chọn Maven và thêm.



Sau khi cài Maven, file **pom.xml** xuất hiện, ta cần thêm như sau để có được driver của MySQL:

<?*xml version*="1.0" *encoding*="UTF-8"?>  
<project *xmlns*="http://maven.apache.org/POM/4.0.0"  
 *xmlns:xsi*="http://www.w3.org/2001/XMLSchema-instance"  
 *xsi:schemaLocation*="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>groupId</groupId>  
 <artifactId>TaoHibernate</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <dependencies>  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>8.0.19</version>  
 </dependency>  
 </dependencies>  
  
  
</project>

## Session

### Định nghĩa và sử dụng

Một Session trong Hibernate được sử dụng để có được một kết nối vật lý với một cơ sở dữ liệu. Đối tượng Session là nhẹ và được thiết kế để được khởi tạo mỗi khi cần tương tác với cơ sở dữ liệu. Các đối tượng persistent được lưu và truy xuất thông qua một đối tượng Session.

Các đối tượng session không nên được mở trong một thời gian dài bởi vì chúng không phải là thread an toàn và nên được tạo ra và đóng khi sau khi sử dụng. Chức năng chính của session là cung cấp các thao tác create, update, read và delete cho các thể hiện các lớp thực thể được ánh xạ. Các thể hiện có thể tồn tại ở một trong ba trạng thái sau đây tại một thời điểm nhất định:

**transient:** Một thể hiện mới của một lớp persistent không liên quan đến một Session, không có đại diện trong cơ sở dữ liệu và không có giá trị định danh được coi là transient (tạm thời) bởi Hibernate.

**persistent:** Bạn có thể tạo một thể hiển transient persistent bằng cách liên kết nó với một session. Một thể hiện persistent có một dại diện trong cơ sở dữ liệu, một giá trị định danh và được liên kết với một Session.

**detached:** Khi chúng ta đóng session trong Hibernate, thể hiện persistent sẽ trở thành một thể hiện detached.

Một thể hiện Session là serializable nếu các lớp persistent của nó là serializable. Một transaction thông thường nên sử dụng cấu trúc sau đây:

Session session = factory.openSession();  
 Transaction tx = *null*;  
  
*try* {  
 tx = session.beginTransaction();  
 *// do some work* ...  
 tx.commit();  
 }  
  
*catch* (Exception e) {  
 *if* (tx!=*null*) tx.rollback();  
 e.printStackTrace();  
 } *finally* {  
 session.close();  
 }

Nếu session ném một ngoại lệ, transaction phải được rollback và session phải là detached.

### Các phương thức của interface Session

Có nhiều phương thức được cung cấp bởi giao tiếp Session nhưng bảng dưới đây chỉ liệt kê vài phương thức quan trọng.

|  |  |
| --- | --- |
| **No.** | **Các phương thức của Session và mô tả** |
| 1 | **Transaction beginTransaction()**  Bắt đầu một đơn vị công việc và trả về đối tượng Transaction liên quan. |
| 2 | **void cancelQuery()**  Hủy việc thực hiện truy vấn hiện tại. |
| 3 | **void clear()**  clear session. |
| 4 | **Connection close()**  Đóng session. |
| 5 | **Criteria createCriteria(Class persistentClass)**  Tạo một thể hiện Criteria, đối với lớp entity đã cho hoặc một superclass của một lớp entity. |
| 6 | **Criteria createCriteria(String entityName)**  Tạo một thể hiện Criteria, đối với entity name đã cho. |
| 7 | **Serializable getIdentifier(Object object)**  Trả lại giá trị số nhận dạng của thực thể cho trước khi kết hợp với session này. |
| 8 | **Query createFilter(Object collection, String queryString)**  Tạo một thể hiện Query mới cho collection và chuỗi lọc nhất định. |
| 9 | **Query createQuery(String queryString)**  Tạo một thể hiện Query mới cho chuỗi truy vấn HQL đã cho. |
| 10 | **SQLQuery createSQLQuery(String queryString)**  Tạo một thể hiện SqlQuery mới cho chuỗi truy vấn SQL đã cho. |
| 11 | **void delete(Object object)**  Xóa persistent instance từ datastore. |
| 12 | **void delete(String entityName, Object object)**  Xóa persistent instance từ datastore. |
| 13 | **Session get(String entityName, Serializable id)**  Trả về persistent instance của thực thể được đặt tên với một số nhận dạng nhất định, hoặc null nếu không có persistent instance. |
| 14 | **SessionFactory getSessionFactory()**  Get session factory cái mà tạo ra session này. |
| 15 | **void refresh(Object object)**  Đọc lại trạng thái của thể hiện đã cho từ cơ sở dữ liệu cơ bản. |
| 16 | **Transaction getTransaction()**  Get thể hiện của Transaction liên quan đến session này. |
| 17 | **boolean isConnected()**  Kiểm tra xem session này có đang kết nối không. |
| 18 | **boolean isDirty()**  Session này có chứa bất kỳ thay đổi nào phải được đồng bộ với cơ sở dữ liệu không? |
| 19 | **boolean isOpen()**  Kiểm tra xem session vẫn đang được open không. |
| 20 | **Serializable save(Object object)**  Lưu transient instance. |
| 21 | **void saveOrUpdate(Object object)**  Thực hiện save(Object) hoặc update(Object). |
| 22 | **void update(Object object)**  Update persistent instance với bộ nhận dạng của detached instance đã cho. |
| 23 | **void update(String entityName, Object object)**  Update persistent instance với bộ nhận dạng của detached instance đã cho. |

## Hibernate - Persistent Class

Toàn bộ ý tưởng của Hibernate là lấy các giá trị từ các thuộc tính(attributes) Java trong Class rồi persist nó vào một bảng database tương ứng. Một mapping document sẽ giúp Hibernate trong việc quyết định kéo các giá trị từ các lớp và ánh xạ chúng với các bảng và trường liên quan.

Các lớp Java có các đối tượng hoặc các thể hiện sẽ được lưu trữ trong các bảng cơ sở dữ liệu được gọi là các **lớp persistent** trong Hibernate. Hibernate hoạt động tốt nhất nếu các lớp này tuân theo một số quy tắc đơn giản, còn được gọi là mô hình lập trình Plain Old Java Object (POJO). Có các quy tắc chính sau đây của các lớp persistent. Tuy nhiên, không có các quy tắc nào là một yêu cầu khó để thực hiện.

* Tất cả các lớp Java persistent cần có constructor mặc định.
* Tất cả các lớp phải có một ID để cho phép dễ dàng xác định các đối tượng của bạn trong Hibernate và cơ sở dữ liệu. Thuộc tính này ánh xạ tới cột khoá chính của một bảng cơ sở dữ liệu (optional).
* Tất cả các thuộc tính phải có phương thức setter và getter.
* Lớp persistent là non-final hoặc được impliments từ interface có các phương thức là public (optional).
* Tất cả các lớp không extend hoặc implement một vài lớp và interface đặc biệt được yêu cầu bởi EJB framework.

Cái tên POJO được sử dụng để nhấn mạnh rằng một đối tượng nhất định là một đối tượng Java thông thường, không phải là một đối tượng đặc biệt, và đặc biệt là một Enterprise JavaBean.

**Đối tượng persistent** có nghĩa là các đối tượng cụ thể có thể sống trong ứng dụng. Chúng có thể được lưu vào một data store và được tái tạo tại một điểm sau này.

### Một ví dụ về Java POJO đơn giản

*public class* Employee {  
 *private int* id;  
 *private* String firstName;  
 *private* String lastName;  
 *private int* salary;  
  
 *public* Employee() {}  
 *public* Employee(String fname, String lname, *int* salary) {  
 *this*.firstName = fname;  
 *this*.lastName = lname;  
 *this*.salary = salary;  
 }  
 *public int* getId() {  
 *return* id;  
 }  
 *public void* setId( *int* id ) {  
 *this*.id = id;  
 }  
 *public* String getFirstName() {  
 *return* firstName;  
 }  
 *public void* setFirstName( String first\_name ) {  
 *this*.firstName = first\_name;  
 }  
 *public* String getLastName() {  
 *return* lastName;  
 }  
 *public void* setLastName( String last\_name ) {  
 *this*.lastName = last\_name;  
 }  
 *public int* getSalary() {  
 *return* salary;  
 }  
 *public void* setSalary( *int* salary ) {  
 *this*.salary = salary;  
 }  
}

## Hibernate - Mapping Files

Ánh xạ đối tượng / quan hệ thường được xác định trong file XML. Tệp ánh xạ này hướng dẫn Hibernate - làm thế nào để ánh xạ lớp (hoặc các lớp) đã xác định vào các bảng cơ sở dữ liệu?

Dù có nhiều người dùng Hibernate chọn tự viết file XML theo cách thủ công, tuy nhiên có một số công cụ cung cấp sẵn để tạo file mapping. Có thể kể đến như: **XDoclet, Middlegen** và **AndroMDA** .

Let us consider our previously defined POJO class whose objects will persist in the table defined in next section

Có hai cách để tạo ra các file mapping XML này, vấn đề này sẽ được đề cập đến trong các bài ví dụ.

* Tạo file mapping XML bằng tay
* Tạo file mapping XML bằng tool, bao gồm XDoclet, Middlegen và AndroMDA

### Ví dụ file mapping XML trong Hiberante

Chúng ta hãy xem xét lớp POJO đã được xác định trước đó có các đối tượng sẽ tồn tại trong bảng được xác định trong phần tiếp theo.

*public class* Student {  
 *private int* id;  
 *private* String name;  
 *//geter, constructor va seter  
  
 public* Student() {  
 }  
  
 *public* Student(*int* id, String name) {  
 *this*.id = id;  
 *this*.name = name;  
 }  
  
 *public int* getId() {  
 *return* id;  
 }  
  
 *public void* setId(*int* id) {  
 *this*.id = id;  
 }  
  
 *public* String getName() {  
 *return* name;  
 }  
  
 *public void* setName(String name) {  
 *this*.name = name;  
 }  
}

Tạo một bảng tương ứng với mỗi đối tượng persistent. Xem xét các đối tượng trên cần phải được lưu trữ và truy xuất vào bảng RDBMS sau:

*CREATE TABLE* `tbl\_student` (  
 `id` *INTEGER NOT NULL*,  
 `name` *VARCHAR*(255) *NOT NULL*,  
 *PRIMARY KEY* (`id`)  
);

Dựa trên hai thực thể trên, chúng ta có thể định nghĩa file mapping XML sau, chỉ dẫn Hibernate làm thế nào để ánh xạ lớp hoặc các lớp đã định nghĩa vào các bảng cơ sở dữ liệu.

<?*xml version*='1.0' *encoding*='utf-8'?>  
<!DOCTYPE *hibernate-mapping* PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  
<hibernate-mapping>  
  
 <class *name*="Entity.Student" *table*="tbl\_student" *schema*="demo">  
 <id *name*="id">  
 <column *name*="id" *sql-type*="int(11)"/>  
 </id>  
 <property *name*="name">  
 <column *name*="name" *sql-type*="varchar(255)"/>  
 </property>  
 </class>  
</hibernate-mapping>

Bạn nên lưu file mapping với định dạng <classname>.hbm.xml. Trong ví dụ trên file mapping được lưu với tên Student.hbm.xml.

### Các thẻ trong file mapping

Chúng ta hãy xem chi tiết hơn về các phần tử của file mapping trong Hibernate như sau:

* Tài liệu mapping là một tài liệu XML có **<hibernate-mapping>** là phần tử gốc chứa tất cả các phần tử <class>.
* Các phần tử **<class>** được sử dụng để định nghĩa ánh xạ cụ thể từ các lớp Java sang các bảng cơ sở dữ liệu. Tên lớp Java được chỉ định sử dụng thuộc tính **name** của phần tử lớp và tên bảng cơ sở dữ liệu được chỉ định sử dụng thuộc tính **table**.
* Phần tử **<meta>** là thành phần tùy chọn và có thể được sử dụng để tạo ra mô tả lớp.
* Phần tử **<id>** ánh xạ thuộc tính ID duy nhất trong lớp tới khóa chính của bảng cơ sở dữ liệu. Thuộc tính **name** của id id đề cập đến thuộc tính trong lớp và thuộc tính **column** đề cập đến cột trong bảng cơ sở dữ liệu. Thuộc tính **type** giữ kiểu ánh xạ hibernate, các kiểu mapping này sẽ chuyển đổi từ kiểu dữ liệu Java sang SQL.
* Phần tử **<generator>** bên trong phần tử id được sử dụng để tự động tạo giá trị cho khóa chính. Thiết lập thuộc tính **class** của phần tử generator được đặt là **native** để cho phép hibernate chọn **identity**, **sequence** hoặc **hilo** để tạo khoá chính tùy thuộc vào khả năng của cơ sở dữ liệu.
* Phần tử **<property>** được sử dụng để ánh xạ một thuộc tính của lớp Java vào một cột trong bảng cơ sở dữ liệu. Thuộc tính **name** của phần tử đề cập đến thuộc tính trong lớp và thuộc tính **column** đề cập đến cột trong bảng cơ sở dữ liệu. Thuộc tính **type** giữ kiểu ánh xạ hibernate, các kiểu mapping này sẽ chuyển đổi từ kiểu dữ liệu Java sang SQL.

Ngoài các phần tử trong ví dụ trên, còn có một vài các thuộc tính và các phần tử khác được sử dụng trong file mapping. Chúng sẽ được đề cập đến trong các bài ví dụ về Hibernate.

## Hibernate - Mapping Types

Khi bạn chuẩn bị tạo tài liệu mapping trong Hibernate, bạn phải ánh xạ các kiểu dữ liệu Java vào các kiểu dữ liệu RDBMS. Các kiểu khai báo và sử dụng trong các file mapping không phải là các kiểu dữ liệu Java; Chúng cũng không phải là các kiểu cơ sở dữ liệu SQL. Các loại này được gọi là các kiểu dữ liệu ánh xạ Hibernate, có thể dịch từ kiểu dữ liệu Java sang SQL và ngược lại.

Trong bài này liệt kê tất cả các kiểu dữ liệu cơ bản (nguyên thủy), date and time, đối tượng, và các mapping dựng sẵn khác.

### Các kiểu dữ liệu nguyên thủy

|  |  |  |
| --- | --- | --- |
| **Mapping type** | **Java type** | **ANSI SQL Type** |
| integer | int hoặc java.lang.Integer | INTEGER |
| long | long hoặc java.lang.Long | BIGINT |
| short | short hoặc java.lang.Short | SMALLINT |
| float | float hoặc java.lang.Float | FLOAT |
| double | double hoặc java.lang.Double | DOUBLE |
| big\_decimal | java.math.BigDecimal | NUMERIC |
| character | java.lang.String | CHAR(1) |
| string | java.lang.String | VARCHAR |
| byte | byte hoặc java.lang.Byte | TINYINT |
| boolean | boolean hoặc java.lang.Boolean | BIT |
| yes/no | boolean hoặc java.lang.Boolean | CHAR(1) ('Y' hoặc 'N') |
| true/false | boolean hoặc java.lang.Boolean | CHAR(1) ('T' hoặc 'F') |

### Các kiểu Date and time

|  |  |  |
| --- | --- | --- |
| **Mapping type** | **Java type** | **ANSI SQL Type** |
| date | java.util.Date hoặc java.sql.Date | DATE |
| time | java.util.Date hoặc java.sql.Time | TIME |
| timestamp | java.util.Date hoặc java.sql.Timestamp | TIMESTAMP |
| calendar | java.util.Calendar | TIMESTAMP |
| calendar\_date | java.util.Calendar | DATE |

### Các kiểu Binary và đối tượng lớn

|  |  |  |
| --- | --- | --- |
| **Mapping type** | **Java type** | **ANSI SQL Type** |
| binary | byte[] | VARBINARY (hoặc BLOB) |
| text | java.lang.String | CLOB |
| serializable | any Java class that implements java.io.Serializable | VARBINARY (hoặc BLOB) |
| clob | java.sql.Clob | CLOB |
| blob | java.sql.Blob | BLOB |

### Các kiểu JDK-related

|  |  |  |
| --- | --- | --- |
| **Mapping type** | **Java type** | **ANSI SQL Type** |
| class | java.lang.Class | VARCHAR |
| locale | java.util.Locale | VARCHAR |
| timezone | java.util.TimeZone | VARCHAR |
| currency | java.util.Currency | VARCHAR |

## Hibernate – Examples

## Hibernate - O/R Mappings

Cho đến nay, chúng ta đã thấy việc mapping O/R cơ bản rất đơn giản bằng cách sử dụng hibernate nhưng có 3 chủ đề O/R mapping trong Hibernate quan trọng nhất mà chúng ta phải tìm hiểu chi tiết. Đó là việc mapping về collections, mapping về liên kết giữa các lớp entity và các Component Mapping.

### Collections Mappings

Nếu một entity hoặc lớp có một tập các giá trị trong một biến Collection cụ thể, thì chúng ta có thể ánh xạ các giá trị đó bằng bất kỳ một trong các interface collections có sẵn trong java. Hibernate có thể persist các thể hiện của **java.util.Map, java.util.Set, java.util.SortedMap, java.util.SortedSet, java.util.List** và bất kỳ mảng nào của các entity hoặc giá trị persistent.

|  |  |
| --- | --- |
| **Kiểu Collection** | **Mapping và môt tả** |
| [java.util.Set](https://viettuts.vn/hibernate/hibernate_set_mapping.htm) | Điều này được ánh xạ với một phần tử <set> và được khởi tạo với java.util.HashSet |
| [java.util.SortedSet](https://viettuts.vn/hibernate/hibernate_sortedset_mapping.htm) | Điều này được ánh xạ với một phần tử <set> và được khởi tạo với java.util.TreeSet. Thuộc tính **sort** để định nghĩa một tập được sắp xếp hoặc không. |
| [java.util.List](https://viettuts.vn/hibernate/hibernate_list_mapping.htm) | Điều này được ánh xạ với một phần tử <list> và được khởi tạo với java.util.ArrayList |
| [java.util.Collection](https://viettuts.vn/hibernate/hibernate_bag_mapping.htm) | Điều này được ánh xạ với một phần tử <bag> or <ibag> và được khởi tạo với java.util.ArrayList |
| [java.util.Map](https://viettuts.vn/hibernate/hibernate_map_mapping.htm) | Điều này được ánh xạ với một phần tử <map> và được khởi tạo với java.util.HashMap |
| [java.util.SortedMap](https://viettuts.vn/hibernate/hibernate_sortedmap_mapping.htm) | Điều này được ánh xạ với một phần tử <map> và được khởi tạo với java.util.TreeMap. Thuộc tính **sort** để định nghĩa một tập được sắp xếp hoặc không. |

Mảng được Hibernate hỗ trợ với <primitive-array> cho các kiểu giá trị Java nguyên thủy và <array> cho các kiểu dữ liệu khác. Tuy nhiên, chúng hiếm khi được sử dụng.

Nếu bạn muốn mapping các kiểu collection do người dùng định nghĩa mà không được Hibernate hỗ trợ trực tiếp, bạn cần phải nói với Hibernate về ngữ nghĩa của các collection tuỳ chỉnh của bạn. Điều này không dễ dàng và không nên sử dụng.

### Association Mappings

Việc ánh xạ các liên kết giữa các lớp entity và các mối quan hệ giữa các bảng chính là linh hồn của ORM. Dưới đây là 4 kiểu liên kết:

|  |  |
| --- | --- |
| **Kiểu Mapping** | **Mô tả** |
| **Many-to-One** | Mapping mối quan hệ many-to-one trong Hibernate |
| **One-to-One** | Mapping mối quan hệ one-to-one trong Hibernate |
| **One-to-Many** | Mapping one-to-many relationship using Hibernate |
| **Many-to-Many** | Mapping mối quan hệ many-to-many trong Hibernate |

### Component Mappings

Trường hợp một lớp Entity có thể có một tham chiếu đến một lớp khác như là một biến thành viên. Nếu lớp được tham chiếu đó không có vòng đời của chính nó và hoàn toàn phụ thuộc vào vòng đời của lớp thực thể sở hữu, lớp được tham chiếu đó được gọi là lớp Component.

Việc ánh xạ Collection của Component kiểu này cũng tương tự như việc ánh xạ một collection bình thường. Chúng ta sẽ thấy hai ánh xạ này cụ thể với các ví dụ.

|  |  |
| --- | --- |
| **Kiểu Mapping** | **Description** |
| **Component Mappings** | Mapping cho một lớp có một tham chiếu đến một lớp khác như là một biến thành viên. Trường hợp này chúng ta sử dụng phần tử **<component>** để thiết lập sự tồn tại của các thuộc khác của lớp của biến tham chiếu bên trong lớp sở hữu nó. |

## Hibernate - Annotations

Ở phần trên, bạn đã thấy cách Hibernate sử dụng file mapping XML để chuyển đổi dữ liệu từ Java POJO sang các bảng cơ sở dữ liệu và ngược lại. Hibernate Annotations là cách mới nhất để định nghĩa các mapping mà không cần sử dụng file XML. Bạn có thể sử dụng các annotation bổ sung hoặc để thay thế siêu dữ mapping XML.

Hibernate Annotations là cách mạnh mẽ để cung cấp siêu dữ liệu cho việc mapping đối tượng và bảng quan hệ. Tất cả các siêu dữ liệu được định nghĩa vào file java POJO cùng với mã này giúp người sử dụng đồng thời hiểu được cấu trúc bảng và POJO trong quá trình phát triển.

Nếu bạn muốn ứng dụng của bạn chuyển sang các ứng dụng ORM tương thích khác của EJB 3, bạn phải sử dụng các annotation để biểu diễn các thông tin mapping nhưng vẫn

### Tạo lớp Entity với các annotation

Như đã đề cập ở trên trong khi làm việc với Hibernate Annotation tất cả các siêu dữ liệu được định nghĩa vào file java POJO cùng với mã này giúp người sử dụng đồng thời hiểu được cấu trúc bảng và lớp POJO trong quá trình phát triển.

Tạo bảng để lưu trữ đối tượng

*CREATE TABLE* `tbl\_student` (  
 `id` *INTEGER NOT NULL AUTO\_INCREMENT*,  
 `name` *VARCHAR*(255) *NOT NULL*,  
 *PRIMARY KEY* (`id`)  
);

Sau đây là việc mapping lớp Student với các annotation để ánh xạ các đối tượng với bảng EMPLOYEE được định nghĩa ở trên:

@Entity  
@Table(name = "tbl\_student", schema = "demo")  
*public class* TblStudentEntity {  
 @Id@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id", nullable = *false*)  
 *private int* id;  
 @Basic@Column(name = "name", nullable = *false*, length = 255)  
 *private* String name;

//Getter&Setter etc…  
  
}

Hibernate nhận ra rằng chú thích @Id nằm trên một trường và giả định rằng các thuộc tính của một đối tượng được truy cập trực tiếp thông qua các trường tại runtime. Nếu bạn đã đặt chú thích @Id vào phương thức getId(), bạn sẽ cho phép truy cập thuộc tính thông qua các phương thức getter và setter của theo mặc định. Do đó, tất cả các chú thích khác cũng được đặt trên cả hai trường hoặc phương thức getter. Dưới đây giải thích các chú thích được sử dụng trong lớp trên.

### @Entity Annotation:

Các chú thích tiêu chuẩn EJB 3 được chứa trong gói javax.persistence, vì vậy trước tiên chúng ta phải import gói này. Thứ hai, chúng ta đã sử dụng chú thích Entity vào lớp Employee đánh dấu lớp này như một Entity Bean, do đó nó phải có một constructor không có đối số mà có kiểu là public.

### @Table Annotation:

Chú thích @Table cho phép bạn chỉ định các chi tiết của bảng sẽ được sử dụng để lưu trữ thực thể trong cơ sở dữ liệu.

Chú thích @Table cung cấp bốn thuộc tính, cho phép bạn ghi đè tên của bảng, danh mục, lược đồ, bắt buộc các ràng buộc duy nhất trên các cột trong bảng. Trong lớp Employee ở trên chúng ta đang sử dụng tên bảng là EMPLOYEE.

### @Id and @GeneratedValue Annotations:

Mỗi entity bean sẽ có một khóa chính, mà bạn chú thích trên lớp với chú thích @Id. Khóa chính có thể là một trường duy nhất hoặc kết hợp nhiều trường tùy thuộc vào cấu trúc bảng của bạn.

Theo mặc định, chú thích @Id sẽ tự động xác định chiến lược tạo primary key, nhưng bạn có thể ghi đè bằng cách áp dụng chú thích **@GeneratedValue** có hai tham số **strategy** và **generator**. Chúng ta chỉ sử dụng chiến lược tạo key mặc định. Cho phép Hibernate xác định loại kiểu tạo nào để sử dụng làm cho mã di chuyển giữa các cơ sở dữ liệu khác nhau.

### @Column Annotation:

Chú thích @Column được sử dụng để chỉ định chi tiết của cột mà trường hoặc thuộc tính sẽ được ánh xạ. Bạn có thể sử dụng chú thích cột với các thuộc tính được sử dụng phổ biến nhất sau đây:

* Thuộc tính **name** được sử dụng để chị định tên cột nào trong db map với tên trường được chú thích.
* Thuộc tính **length** cho phép kích thước của cột được sử dụng để ánh xạ một giá trị đặc biệt cho một giá trị String.
* Thuộc tính nullable cho phép cột được đánh dấu KHÔNG NULL khi schema được tạo ra.
* Thuộc tính **unique** cho phép cột được đánh dấu chỉ chứa các giá trị duy nhất.

## Hibernate - Query Language

Hibernate Query Language (HQL) là một ngôn ngữ truy vấn hướng đối tượng, tương tự như SQL nhưng thay vì làm việc trên các bảng và cột, HQL làm việc với các đối tượng persistent và các thuộc tính của chúng. Các truy vấn HQL được dịch bởi Hibernate thành các truy vấn SQL thông thường, lần lượt thực hiện các công việc trên cơ sở dữ liệu.

Mặc dù bạn có thể sử dụng các câu lệnh SQL trực tiếp với Hibernate bằng cách sử dụng [Native SQL](https://viettuts.vn/hibernate/native-sql-trong-hibernate). Nhưng chúng tôi khuyên bạn nên sử dụng HQL bất cứ khi nào có thể để tránh sự phức tạp về tính linh hoạt của cơ sở dữ liệu và tận dụng chiến lược generation và caching của Hibernate.

Các từ khoá như SELECT, FROM và WHERE vv không phân biệt chữ hoa và chữ thường nhưng các thuộc tính như tên bảng và cột có phân biệt chữ hoa chữ thường trong HQL.

### Mệnh đề FROM

Bạn sẽ sử dụng mệnh đề **FROM** nếu bạn muốn tải các các đối tượng persistent vào bộ nhớ. Sau đây là cú pháp đơn giản sử dụng mệnh đề FROM:

String hql = "FROM Employee";  
Query query = session.createQuery(hql);  
List results = query.list();

Nếu bạn cần sử dụng tên lớp đầy đủ trong HQL, chỉ cần chỉ định gói và tên lớp như sau:

String hql = "FROM com.hibernatebook.criteria.Employee";  
Query query = session.createQuery(hql);  
List results = query.list();

### Mệnh đề AS

Mệnh đề **AS** có thể được sử dụng để gán các bí danh cho các lớp trong các truy vấn HQL, đặc biệt khi bạn có các truy vấn dài. Ví dụ:

String hql = "FROM Employee AS E";  
Query query = session.createQuery(hql);  
List results = query.list();

Từ khoá **AS** là tùy chọn và bạn cũng có thể chỉ định bí danh trực tiếp sau tên lớp, như sau:

String hql = "FROM Employee E";  
Query query = session.createQuery(hql);  
List results = query.list();

### Mệnh đề SELECT

Mệnh đề SELECT cung cấp kiểm soát nhiều hơn các tập kết quả hơn mệnh đề from. Nếu bạn muốn lấy ra những thuộc tính cần thiết của các đối tượng thay vì đối tượng hoàn chỉnh, hãy sử dụng mệnh đề SELECT. Sau đây là cú pháp đơn giản của việc sử dụng mệnh đề SELECT để chỉ lấy ra trường first\_name của đối tượng Employee:

String hql = "SELECT E.firstName FROM Employee E";  
Query query = session.createQuery(hql);  
List results = query.list();

Chú ý: **Employee.firstName** là thuộc tính của đối tượng Employee chứ không phải là một trường của bảng EMPLOYEE...

### Mệnh đề WHERE

Nếu bạn muốn thu hẹp các đối tượng cụ thể được trả về từ cơ sở dữ liệu, bạn sử dụng mệnh đề WHERE. Sau đây là cú pháp đơn giản sử dụng mệnh đề WHERE:

String hql = "FROM Employee E WHERE E.id = 10";  
Query query = session.createQuery(hql);  
List results = query.list();

### Mệnh đề ORDER BY

Để sắp xếp kết quả truy vấn HQL của bạn, bạn sẽ cần sử dụng mệnh đề **ORDER BY**. Bạn có thể sắp xếp các kết quả theo bất kỳ thuộc tính nào trên các đối tượng trong tập kết quả hoặc tăng dần (ASC) hoặc giảm dần (DESC). Sau đây là cú pháp đơn giản của việc sử dụng mệnh đề ORDER BY:

String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";  
Query query = session.createQuery(hql);  
List results = query.list();

Nếu bạn muốn sắp xếp theo nhiều thuộc tính, bạn chỉ cần thêm các thuộc tính bổ sung vào cuối của mệnh đề ORDER BY, được phân cách bằng dấu phẩy như sau:

String hql = "FROM Employee E WHERE E.id > 10 " +  
 "ORDER BY E.firstName DESC, E.salary DESC ";  
Query query = session.createQuery(hql);  
List results = query.list();

### Mệnh đề GROUP BY

Mệnh đề này cho phép Hibernate lấy thông tin từ cơ sở dữ liệu và phân nhóm chúng dựa trên giá trị của một thuộc tính. Sau đây là cú pháp đơn giản của việc sử dụng mệnh đề GROUP BY:

String hql = "SELECT SUM(E.salary), E.firtName FROM Employee E " +  
 "GROUP BY E.firstName";  
Query query = session.createQuery(hql);  
List results = query.list();

### Sử dụng các tham số đặt tên

Hibernate hỗ trợ các tham số được đặt tên trong các truy vấn HQL của nó. Điều này làm cho việc viết các truy vấn HQL chấp nhận đầu vào từ người dùng dễ dàng và bạn không phải phòng chống các cuộc tấn công SQL injection. Sau đây là cú pháp đơn giản sử dụng các tham số được đặt tên:

String hql = "FROM Employee E WHERE E.id = :employee\_id";  
 Query query = session.createQuery(hql);  
query.setParameter("employee\_id",10);  
 List results = query.list();

### Mệnh đề UPDATE

Cập nhật hàng loạt là kiểu mới đối với HQL trong Hibernate 3. Và xóa khác nhau trong Hibernate 3 so với Hibernate 2. Giao tiếp Query giờ đây chứa một phương thức gọi là executeUpdate() để thực hiện các câu lệnh HQL UPDATE hoặc DELETE.

Mệnh đề **UPDATE** có thể được sử dụng để cập nhật một hoặc nhiều thuộc tính của một hoặc nhiều đối tượng. Sau đây là cú pháp đơn giản sử dụng mệnh đề UPDATE:

String hql = "UPDATE Employee set salary = :salary " +  
 "WHERE id = :employee\_id";  
 Query query = session.createQuery(hql);  
 query.setParameter("salary", 1000);  
 query.setParameter("employee\_id", 10);  
 *int* result = query.executeUpdate();  
 System.out.println("Rows affected: " + result);

### Mệnh đề DELETE

Mệnh đề **DELETE** được sử dụng để xóa một hoặc nhiều đối tượng. Sau đây là cú pháp đơn giản sử dụng mệnh đề DELETE:

String hql = "DELETE FROM Employee " +  
 "WHERE id = :employee\_id";  
Query query = session.createQuery(hql);  
query.setParameter("employee\_id", 10);  
*int* result = query.executeUpdate();  
System.out.println("Rows affected: " + result);

### Mệnh đề INSERT

HQL chỉ hỗ trợ mệnh đề INSERT INTO nơi mà các bản ghi có thể được insert từ một đối tượng này sang một đối tượng khác. Sau đây là cú pháp đơn giản sử dụng mệnh đề INSERT INTO:

String hql = "INSERT INTO Employee(firstName, lastName, salary)" +  
 "SELECT firstName, lastName, salary FROM old\_employee";  
 Query query = session.createQuery(hql);  
 *int* result = query.executeUpdate();  
System.out.println("Rows affected: " + result);

### Các phương thức tổng hợp

HQL hỗ trợ một loạt các phương thức tổng hợp tương tự như SQL. Chúng hoạt động theo cùng một cách trong HQL như trong SQL và sau đây là danh sách các chức năng có sẵn:

|  |  |  |
| --- | --- | --- |
| **No.** | **Các hàm** | **Mô tả** |
| 1 | **avg(property name)** | Trung bình của các giá trị của một thuộc tính. |
| 2 | **count(property name or \*)** | Số lần một thuộc tính xảy ra trong kết quả. |
| 3 | **max(property name)** | Giá tri lớn nhất trong các giá trị của một trường. |
| 4 | **min(property name)** | The minimum value of the property values |
| 5 | **sum(property name)** | Tính tổng các giá trị của một trường |

Từ khóa **distinct** chỉ tính các giá trị duy nhất trong tập hàng. Truy vấn sau đây sẽ chỉ trả về số lần duy nhất:

String hql = "SELECT count(distinct E.firstName) FROM Employee E";  
Query query = session.createQuery(hql);  
List results = query.list();

### Phân trang bằng việc sử dụng Query

Có hai phương thức của giao tiếp Query cho việc phân trang.

|  |  |
| --- | --- |
| **No.** | **Phương thức & Mô tả** |
| 1 | **Query setFirstResult(int startPosition)**  Phương thức này lấy một số nguyên đại diện cho hàng đầu tiên trong tập kết quả, bắt đầu với hàng 0. |
| 2 | **Query setMaxResults(int maxResult)**  Phương thức này cho Hibernate tìm kiếm một số **maxResults**của các đối tượng. |

Sử dụng hai phương thức trên, chúng ta có thể xây dựng một thành phần phân trang trong ứng dụng Web hoặc Swing. Sau đây là ví dụ bạn có thể mở rộng để lấy 10 hàng cùng một lúc:

String hql = "FROM Employee";  
 Query query = session.createQuery(hql);  
 query.setFirstResult(1);  
 query.setMaxResults(10);  
 List results = query.list();

## Hibernate - Criteria Queries

Hibenate cung cấp Criteria API chứa các phương thức cho phép bạn xây dựng một đối tượng truy vấn tiêu chuẩn theo yêu cầu của chương trình, nơi bạn có thể áp dụng các quy tắc lọc và điều kiện hợp lý.

Giao tiếp Session trong Hibernate cung cấp phương thức createCriteria() có thể được sử dụng để tạo một đối tượng Criteria mà trả về các thể hiện của lớp đối tượng persistence khi ứng dụng của bạn thi hành một truy vấn tiêu chuẩn.

Sau đây là ví dụ đơn giản nhất của một truy vấn tiêu chuẩn, chỉ đơn giản là trả về mọi đối tượng tương ứng với lớp Employee.

*Criteria* cr = session.createCriteria(Employee.*class*);  
*List* results = cr.list();

### Restrictions với Criteria

Bạn có thể sử dụng phương thức **add()** có sẵn cho đối tượng Criteria để thêm các điều kiện lọc cho truy vấn tiêu chuẩn. Sau đây là ví dụ để bổ sung điều kiện để trả các record có salary là 2000:

*Criteria* cr = session.createCriteria(Employee.*class*);  
cr.add(Restrictions.eq("salary", 2000));  
*List* results = cr.list();

Sau đây là một vài ví dụ khác bao gồm các kịch bản khác nhau và có thể được sử dụng theo từng yêu cầu:

*Criteria* cr = session.createCriteria(Employee.*class*);  
  
*// Lấy các record có salary lớn hơn 2000*cr.add(Restrictions.gt("salary", 2000));  
  
*// Lấy các record có salary nhỏ hơn 2000*cr.add(Restrictions.lt("salary", 2000));  
  
*// Lấy các record có fistName bắt đầu là zara*cr.add(Restrictions.like("firstName", "zara%"));  
  
*// Dạng không phân biệt chữ hoa và chữ thường cho điều kiện trên.*cr.add(Restrictions.ilike("firstName", "zara%"));  
  
*// Lấy các record có salary trong khoảng 1000 và 2000*cr.add(Restrictions.between("salary", 1000, 2000));  
  
*// Check thuộc tính đã cho là null*cr.add(Restrictions.isNull("salary"));  
  
*// Check thuộc tính đã cho là khác null*cr.add(Restrictions.isNotNull("salary"));  
  
*// Check thuộc tính đã cho là empty*cr.add(Restrictions.isEmpty("salary"));  
  
*// Check thuộc tính đã cho là khác empty*cr.add(Restrictions.isNotEmpty("salary"));

Bạn có thể tạo các điều kiện AND và OR sử dụng các hạn chế LogicExpression như sau:

*Criteria* cr = session.createCriteria(Employee.*class*);  
  
 *Criterion* salary = Restrictions.*gt*("salary", 2000);  
 *Criterion* name = Restrictions.*ilike*("firstNname","zara%");  
  
 *// Lấy các record phù hợp với điều kiện OR* LogicalExpression orExp = Restrictions.*or*(salary, name);  
 cr.add(orExp);  
  
 *// Lấy các record phù hợp với điều kiện AND* LogicalExpression andExp = Restrictions.*and*(salary, name);  
 cr.add(andExp);  
  
 *List* results = cr.list();

Tuy nhiên tất cả các điều kiện trên có thể được sử dụng trực tiếp với HQL như đã giải thích trong hướng dẫn trước.

### Phân trang bằng việc sử dụng Criteria

Có hai phương thức của giao tiếp Criteria để phân trang.

|  |  |
| --- | --- |
| **No.** | **Phương thức & Mô tả** |
| 1 | **public Criteria setFirstResult(int firstResult)**  Phương thức này lấy một số nguyên đại diện cho hàng đầu tiên trong tập kết quả của bạn, bắt đầu với hàng 0. |
| 2 | **public Criteria setMaxResults(int maxResults)**  Phương thức này cho Hibernate tìm kiếm một số **maxResults của các đối tượng.** |

Sử dụng hai phương thức trên, chúng ta có thể xây dựng một thành phần phân trang trong ứng dụng Web hoặc Swing. Sau đây là ví dụ mà bạn có thể mở rộng để lấy 10 hàng một lúc:

*Criteria* cr = session.createCriteria(Employee.*class*);  
cr.setFirstResult(1);  
cr.setMaxResults(10);  
*List* results = cr.list();

### Sắp xếp kết quả

Criteria API cung cấp lớp org.hibernate.criterion.Order để sắp xếp kết quả trả về theo thứ tự tăng dần hoặc giảm dần, theo một trong các thuộc tính của đối tượng. Ví dụ này thể hiện cách sử dụng lớp Order để sắp xếp tập kết quả:

*Criteria* cr = session.createCriteria(Employee.*class*);  
  
*// Lấy các record có salary lớn hơn 2000*cr.add(Restrictions.gt("salary", 2000));  
  
*// Sắp xếp các record theo thứ tự giảm dần*cr.addOrder(Order.desc("salary"));  
  
*// Sắp xếp các record theo thứ tự tăng dần*cr.addOrder(Order.asc("salary"));  
  
 *List* results = cr.list();

### Projections & Aggregations

Criteria API cung cấp lớp org.hibernate.criterion.Projections có thể được sử dụng để có được giá trị trung bình, maximum và minimum của các giá trị thuộc tính. Lớp Projections cũng tương tự như lớp Restrictions vì nó cung cấp một số phương thức static để lấy các thể hiện của lớp Projections.

Dưới đây là một số ví dụ về các trường hợp khác nhau và có thể được sử dụng theo từng yêu cầu:

*Criteria* cr = session.createCriteria(Employee.*class*);  
  
*// Lấy tổng số hàng.*cr.setProjection(Projections.rowCount());  
  
*// Lấy giá trị trung bình của một thuộc tính.*cr.setProjection(Projections.avg("salary"));  
  
*// Để có số lần xuất hiện duy nhât của một thuộc tính.*cr.setProjection(Projections.countDistinct("firstName"));  
  
*// Lấy giá trị maximum của một thuộc tính.*cr.setProjection(Projections.max("salary"));  
  
*// Lấy giá trị minimum của một thuộc tính.*cr.setProjection(Projections.min("salary"));  
  
*// Lấy tổng các giá trị của một thuộc tính*cr.setProjection(Projections.sum("salary"));

### Ví dụ về câu truy vấn Criteria

Tạo lớp Java POJO Employee.java

*public class* Employee {  
 *private int* id;  
 *private* String firstName;  
 *private* String lastName;  
 *private int* salary;  
  
 *public* Employee() {}  
 *public* Employee(String fname, String lname, *int* salary) {  
 *this*.firstName = fname;  
 *this*.lastName = lname;  
 *this*.salary = salary;  
 }  
 *public int* getId() {  
 *return* id;  
 }  
 *public void* setId( *int* id ) {  
 *this*.id = id;  
 }  
 *public* String getFirstName() {  
 *return* firstName;  
 }  
 *public void* setFirstName( String first\_name ) {  
 *this*.firstName = first\_name;  
 }  
 *public* String getLastName() {  
 *return* lastName;  
 }  
 *public void* setLastName( String last\_name ) {  
 *this*.lastName = last\_name;  
 }  
 *public int* getSalary() {  
 *return* salary;  
 }  
 *public void* setSalary( *int* salary ) {  
 *this*.salary = salary;  
 }

Tạo bảng EMPLOYEE sau để lưu trữ các đối tượng Employee

*create table* EMPLOYEE (  
 id *INT NOT NULL auto\_increment*,  
 first\_name *VARCHAR*(20) *default NULL*,  
 last\_name *VARCHAR*(20) *default NULL*,  
 salary *INT default NULL*,  
 *PRIMARY KEY* (id)  
);

Tạo file mapping XML

File: **Employee.hbm.xml**

<?*xml version*="1.0" *encoding*="utf-8"?>  
<!DOCTYPE *hibernate-mapping* PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  
  
<hibernate-mapping>  
 <class *name*="Employee" *table*="EMPLOYEE">  
 <meta *attribute*="class-description">  
 Lớp này chưa thông tin chi tiết về employee.  
 </meta>  
 <id *name*="id" *type*="int" *column*="id">  
 <generator *class*="native" />  
 </id>  
 <property *name*="firstName" *column*="first\_name" *type*="string" />  
 <property *name*="lastName" *column*="last\_name" *type*="string" />  
 <property *name*="salary" *column*="salary" *type*="int" />  
 </class>  
</hibernate-mapping>

Tạo lớp ứng dụng chứa phương thức main() để chạy ứng dụng nơi mà truy vấn Criteria được sử dụng

*public class* ManageEmployee {  
 *private static* SessionFactory *factory*;  
  
 *public static void* main(String[] args) {  
 *try*{  
 *factory* = *new* Configuration().configure().buildSessionFactory();  
 }*catch* (Throwable ex) {  
 System.err.println("Failed to create sessionFactory object." + ex);  
 *throw new* ExceptionInInitializerError(ex);  
 }  
 ManageEmployee ME = *new* ManageEmployee();  
  
 *// Add few employee records in database* Integer empID1 = ME.addEmployee("Zara", "Ali", 2000);  
 Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);  
 Integer empID3 = ME.addEmployee("John", "Paul", 5000);  
 Integer empID4 = ME.addEmployee("Mohd", "Yasee", 3000);  
  
 *// List down all the employees* ME.listEmployees();  
  
 *// Print Total employee's count* ME.countEmployee();  
  
 *// Print Toatl salary* ME.totalSalary();  
 }  
  
 *// Method to CREATE an employee in the database  
 public* Integer addEmployee(String fname, String lname, *int* salary){  
 Session session = *factory*.openSession();  
 Transaction tx = *null*;  
 Integer employeeID = *null*;  
 *try*{  
 tx = session.beginTransaction();  
 Employee employee = *new* Employee(fname, lname, salary);  
 employeeID = (Integer) session.save(employee);  
 tx.commit();  
 }*catch* (HibernateException e) {  
 *if* (tx!=*null*) tx.rollback();  
 e.printStackTrace();  
 }*finally* {  
 session.close();  
 }  
 *return* employeeID;  
 }  
  
 *// Method to READ all the employees having salary more than 2000  
 public void* listEmployees( ){  
 Session session = *factory*.openSession();  
 Transaction tx = *null*;  
 *try*{  
 tx = session.beginTransaction();  
 *Criteria* cr = session.createCriteria(Employee.*class*);  
 *// Add restriction.* cr.add(Restrictions.*gt*("salary", 2000));  
 *List* employees = cr.list();  
  
 *for* (*Iterator* iterator = employees.iterator(); iterator.hasNext();){  
 Employee employee = (Employee) iterator.next();  
 System.out.print("First Name: " + employee.getFirstName());  
 System.out.print(" Last Name: " + employee.getLastName());  
 System.out.println(" Salary: " + employee.getSalary());  
 }  
 tx.commit();  
 }*catch* (HibernateException e) {  
 *if* (tx!=*null*) tx.rollback();  
 e.printStackTrace();  
 }*finally* {  
 session.close();  
 }  
 }  
 *// Method to print total number of records  
 public void* countEmployee(){  
 Session session = *factory*.openSession();  
 Transaction tx = *null*;  
 *try*{  
 tx = session.beginTransaction();  
 *Criteria* cr = session.createCriteria(Employee.*class*);  
  
 *// To get total row count.* cr.setProjection(Projections.*rowCount*());  
 *List* rowCount = cr.list();  
  
 System.out.println("Total Coint: " + rowCount.get(0) );  
 tx.commit();  
 }*catch* (HibernateException e) {  
 *if* (tx!=*null*) tx.rollback();  
 e.printStackTrace();  
 }*finally* {  
 session.close();  
 }  
 }  
 *// Method to print sum of salaries  
 public void* totalSalary(){  
 Session session = *factory*.openSession();  
 Transaction tx = *null*;  
 *try*{  
 tx = session.beginTransaction();  
 *Criteria* cr = session.createCriteria(Employee.*class*);  
  
 *// To get total salary.* cr.setProjection(Projections.*sum*("salary"));  
 *List* totalSalary = cr.list();  
  
 System.out.println("Total Salary: " + totalSalary.get(0) );  
 tx.commit();  
 }*catch* (HibernateException e) {  
 *if* (tx!=*null*) tx.rollback();  
 e.printStackTrace();  
 }*finally* {  
 session.close();  
 }  
 }  
}

Kết quả chạy:

First Name: Daisy Last Name: Das Salary: 5000

First Name: John Last Name: Paul Salary: 5000

First Name: Mohd Last Name: Yasee Salary: 3000

Total Coint: 4

Total Salary: 15000

## Hibernate - Native SQL

Bạn có thể sử dụng Native SQL trong Hibernate để thực hiện các truy vấn cơ sở dữ liệu nếu bạn muốn sử dụng các tính năng cơ sở dữ liệu cụ thể như hint hoặc từ khoá CONNECT trong Oracle. Hibernate 3.x cho phép bạn chỉ định SQL viết tay, bao gồm các stored procedures, và tất cả các thao tác create, update, delete và select.

Bạn có thể tạo một truy vấn Native SQL từ session với phương thức createSQLQuery() trên interface Session:

*public SQLQuery* createSQLQuery(String sqlString) *throws* HibernateException

## Các kiểu truy vấn Native SQL

Sau khi bạn truyền một chuỗi chứa truy vấn SQL đến phương thức createSQLQuery(), bạn phải liên kết kết quả trả về SQL cho thực thể Hibernate hiện có, một join, hoặc một kết quả vô hướng (scalar) sử dụng các phương thức addEntity(), addJoin() và addScalar() tương ứng.

### 1. Truy vấn vô hướng (Scalar)

Các truy vấn SQL cơ bản nhất là để có được một danh sách các vô hướng (giá trị) từ một hoặc nhiều bảng. Sau đây là cú pháp để sử dụng SQL gốc cho các giá trị vô hướng:

String sql = "SELECT first\_name, salary FROM EMPLOYEE";  
 *Query* query = session.createSQLQuery(sql);  
 query.setResultTransformer(*Criteria*.ALIAS\_TO\_ENTITY\_MAP);  
 *List* results = query.list();

### 2. Truy vấn Entity

Các truy vấn ở trên là về trả về các giá trị vô hướng từ resultset. Sau đây là cú pháp để có được các đối tượng Entity từ một truy vấn Native SQL thông qua addEntity().

String sql = "SELECT \* FROM EMPLOYEE";  
Query query = session.createSQLQuery(sql);  
query.addEntity(Employee.*class*);  
*List* results = query.list();

**Truy vấn Entity với Named SQL**

Sau đây là cú pháp để nhận các đối tượng thực thể từ truy vấn native sql thông qua addEntity() và sử dụng truy vấn Named SQL.

String sql = "SELECT \* FROM EMPLOYEE WHERE id = :employee\_id";  
 Query query = session.createSQLQuery(sql);  
query.addEntity(Employee.*class*);  
query.setParameter("employee\_id", 10);  
 *List* results = query.list();

### Ví dụ về Native SQL trong Hibernate

Lớp Entity Student:

@Entity  
@Table(name = "tbl\_student", schema = "demo")  
*public class* TblStudentEntity {  
 @Id@GeneratedValue(strategy = GenerationType.IDENTITY)@Column(name = "id", nullable = *false*)  
 *private int* id;  
 @Basic@Column(name = "name", nullable = *false*, length = 255)  
 *private* String name;

//Constructor, Getter & Setter  
}

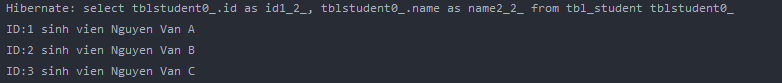
Lớp này sẽ tạo ra bảng sau trong cơ sở dữ liệu:

*create table* tbl\_teacher  
(  
 id *int auto\_increment  
 primary key*,  
 *name varchar*(255) *not null*)

Truy vấn với SQL:

*Session* session = factory.getCurrentSession();  
session.getTransaction().begin();  
String sql = "Select s from "+ TblStudentEntity.*class*.getName() + " s";  
*Query*<TblStudentEntity> query = session.createQuery(sql);  
*List*<TblStudentEntity> studentEntities = query.getResultList();  
*for* (TblStudentEntity student :  
 studentEntities) {  
 System.out.println("ID:" + student.getId() + " sinh vien "+ student.getName());  
}  
session.getTransaction().commit();

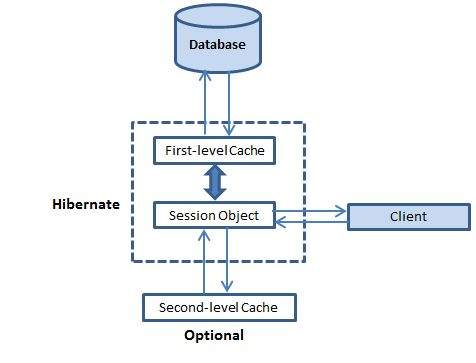
Kết quả:



## Hibernate - Caching

Bộ nhớ đệm là tất cả về tối ưu hóa hiệu của suất ứng dụng. Nó nằm giữa ứng dụng và cơ sở dữ liệu để tránh số lượng lượt truy cập cơ sở dữ liệu càng nhiều càng tốt để cung cấp cho một hiệu suất tốt hơn cho các ứng dụng.

Bộ nhớ đệm cũng rất quan trọng đối với Hibernate sử dụng các lược đồ bộ nhớ đệm nhiều cấp như được giải thích dưới đây:



### Các kiểu bộ nhớ cache trong Hibernate

Có 3 kiểu bộ nhớ cache trong Hibernate đó là: Bộ nhớ cache cấp một (First-level), Bộ nhớ cache cấp hai (Second-level) và Bộ nhớ cache cấp truy vấn.

#### Bộ nhớ cache cấp một (First-level)

Bộ nhớ cache cấp một là bộ nhớ cache Session và là một bộ nhớ cache bắt buộc thông qua đó tất cả các yêu cầu phải vượt qua. Đối tượng Session giữ một đối tượng thuộc quyền sở hữu của nó trước khi commit nó vào cơ sở dữ liệu.

Nếu bạn phát hành nhiều bản cập nhật cho một đối tượng, Hibernate cố gắng trì hoãn việc cập nhật càng lâu càng tốt để giảm số lượng các câu lệnh SQL cập nhật đã ban hành. Nếu bạn đóng session, tất cả các đối tượng được lưu trữ trong bộ nhớ cache đều bị mất và vẫn tiếp tục tồn tại hoặc cập nhật trong cơ sở dữ liệu.

#### Bộ nhớ cache cấp hai (Second-level)

Bộ nhớ cache cấp hai là một bộ nhớ cache tùy chọn. Và bộ nhớ cache cấp một sẽ luôn được thảo luận trước khi thực hiện xác định vị trí một đối tượng trong bộ nhớ cache cấp hai. Bộ nhớ cache cấp hai có thể được cấu hình trên cơ sở mỗi lớp và mỗi collection và chịu trách nhiệm chính trong việc lưu trữ các đối tượng trong các session.

Bộ nhớ cache của bên thứ ba có thể được sử dụng với Hibernate. Một giao tiếp org.hibernate.cache.CacheProvider được cung cấp, phải được implements để cung cấp Hibernate với một xử lý để cài đăt bộ nhớ cache.

#### Bộ nhớ cache cấp truy vấn

Hibernate cũng thực hiện bộ nhớ cache cho các kết quả truy vấn kết hợp chặt chẽ với bộ nhớ cache cấp hai.

Đây là tính năng tùy chọn và yêu cầu thêm hai vùng bộ nhớ cache vật lý giữ kết quả truy vấn và các dấu thời gian khi một bảng được cập nhật lần cuối. Điều này chỉ hữu ích cho các truy vấn được chạy thường xuyên với các tham số giống nhau.

#### Sử dụng bộ nhớ cache trong Hibernate

Hibernate sử dụng bộ nhớ cache cấp một theo mặc định và bạn không phải làm gì để sử dụng bộ nhớ cache cấp một. Hãy đi thẳng đến bộ nhớ cache cấp hai tùy chọn.

#### Bộ nhớ cache cấp hai (Second-level)

Không phải tất cả các lớp đều được hưởng lợi từ bộ nhớ đệm, vì vậy điều quan trọng là có thể vô hiệu bộ nhớ cache cấp hai.

Bộ nhớ cache cấp hai Hibernate được thiết lập trong hai bước. Trước hết, bạn phải quyết định chiến lược truy cập đồng thời nào để sử dụng. Sau đó, bạn cấu hình hết hạn của bộ nhớ cache và thuộc tính bộ nhớ cache vật lý bằng cách sử dụng bộ nhớ cache provider.

### Chiến lược truy cập đồng thời

Chiến lược truy cập đồng thời là bộ điều chỉnh có trách nhiệm lưu trữ các mục dữ liệu trong bộ nhớ cache và lấy chúng từ bộ nhớ cache. Nếu muốn kích hoạt bộ nhớ cache cấp hai, bạn sẽ phải quyết định, đối với mỗi lớp và collection persistent, mà chiến lược truy cập đồng thời vào bộ nhớ cache để sử dụng.

* **Transactional**: Sử dụng chiến lược này để đọc - chủ yếu là dữ liệu, nơi mà nó là quan trọng để ngăn ngừa dữ liệu cũ trong các transaction đồng thời, trong trường hợp hiếm hoi của một cập nhật.
* **Read-write**: Một lần nữa sử dụng chiến lược này để đọc - chủ yếu là dữ liệu, nơi nó là quan trọng để ngăn ngừa dữ liệu cũ trong các transaction đồng thời, trong trường hợp hiếm hoi của một cập nhật.
* **Nonstrict-read-write**: Chiến lược này không đảm bảo tính nhất quán giữa bộ nhớ cache và cơ sở dữ liệu. Sử dụng chiến lược này nếu dữ liệu hầu như không thay đổi và một khả năng nhỏ dữ liệu cũ không phải là mối quan tâm quan trọng.
* **Read-only**: Một chiến lược truy cập đồng thời phù hợp với dữ liệu mà không bao giờ thay đổi. Chỉ sử dụng nó cho dữ liệu tham khảo.

Nếu chúng ta sử dụng bộ nhớ đệm thứ hai cho lớp Employee. Hãy thêm phần tử mapping cần thiết để nói với Hibernate để cache các thể hiện Employee sử dụng chiến lược Read-only.

<?*xml version* = "1.0" *encoding* = "utf-8"?>  
<!DOCTYPE *hibernate-mapping* PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  
<hibernate-mapping>  
 <class *name* = "Employee" *table* = "EMPLOYEE">  
  
 <meta *attribute* = "class-description">  
 This class contains the employee detail.  
 </meta>  
  
 <cache *usage* = "read-write"/>  
  
 <id *name* = "id" *type* = "int" *column* = "id">  
 <generator *class*="native"/>  
 </id>  
  
 <property *name* = "firstName" *column* = "first\_name" *type* = "string"/>  
 <property *name* = "lastName" *column* = "last\_name" *type* = "string"/>  
 <property *name* = "salary" *column* = "salary" *type* = "int"/>  
  
 </class>  
</hibernate-mapping>

Thuộc tính usage = "read-write" cho Hibernate sử dụng một chiến lược truy cập đồng thời đọc-ghi cho bộ nhớ cache được xác định.

### Cache provider

Bước tiếp theo của bạn sau khi lựa chọn chiến lược đồng thời bạn sẽ sử dụng cho các lớp ứng cử viên bộ nhớ cache là việc chọn lựa một cache provider. Hibernate buộc bạn phải chọn một cache provider duy nhất cho toàn bộ ứng dụng.

|  |  |  |
| --- | --- | --- |
| **No.** | **Cache Provider** | **Mô tả** |
| 1 | EHCache | Nó có thể cache trong bộ nhớ RAM hoặc trên đĩa cứng và clustered caching và nó hỗ trợ bộ nhớ cache kết quả truy vấn Hibernate tuỳ chọn. |
| 2 | OSCache | Hỗ trợ bộ nhớ đệm vào bộ nhớ RAM và đĩa cứng trong một JVM duy nhất, với một tập hợp đầy đủ các chính sách hết hạn và hỗ trợ bộ nhớ truy vấn. |
| 3 | warmCache | Một bộ nhớ cache cluster dựa trên JGroups. Nó sử dụng huỷ bỏ hiệu lực clustered nhưng không hỗ trợ bộ nhớ cache truy vấn Hibernate |
| 4 | JBoss Cache | Một bộ nhớ cache cluster được sao chép hoàn toàn hợp lệ được transaction dựa trên thư viện đa nhóm JGroups. Nó hỗ trợ nhân bản hoặc hủy bỏ hiệu lực, giao tiếp đồng bộ hoặc không đồng bộ, và khóa lạc quan và bi quan. Bộ nhớ truy vấn cache Hibernate được hỗ trợ |

Mỗi cache provider không tương thích với mọi chiến lược truy cập đồng thời. Ma trận tương thích sau sẽ giúp bạn chọn một sự kết hợp thích hợp.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strategy/Provider** | **Read-only** | **Nonstrictread-write** | **Read-write** | **Transactional** |
| EHCache | X | X | X |  |
| OSCache | X | X | X |  |
| SwarmCache | X | X |  |  |
| JBoss Cache | X |  |  | X |

Bạn sẽ chỉ định cache provider trong file cấu hình hibernate.cfg.xml. Ví dụ sau, chúng ta chọn EHCache làm cache provider cấp hai:

<?*xml version*="1.0" *encoding*="utf-8"?>  
<!DOCTYPE *hibernate-configuration* SYSTEM  
 "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  
  
<hibernate-configuration>  
 <session-factory>  
 <property *name*="hibernate.dialect">  
 org.hibernate.dialect.MySQLDialect  
 </property>  
 <property *name*="hibernate.connection.driver\_class">  
 com.mysql.jdbc.Driver  
 </property>  
  
 *<!-- thông tin kết nối db -->* <property *name*="hibernate.connection.url">  
 jdbc:mysql://localhost/test  
 </property>  
 <property *name*="hibernate.connection.username">  
 root  
 </property>  
 <property *name*="hibernate.connection.password">  
 root123  
 </property>  
  
 *<!-- cài đặt cache provider -->* <property *name*="hibernate.cache.provider\_class">  
 org.hibernate.cache.EhCacheProvider  
 </property>  
  
 *<!-- List các file XML mapping -->* <mapping *resource*="Employee.hbm.xml"/>  
  
 </session-factory>  
</hibernate-configuration>

Bây giờ, bạn cần chỉ định các thuộc tính của vùng nhớ cache. EHCache có file cấu hình riêng của mình, đó là **ehcache.xml** nên để trong CLASSPATH của ứng dụng. Cấu hình bộ nhớ cache trong ehcache.xml cho lớp Employee có thể như sau:

<diskStore *path*="java.io.tmpdir"/>  
<defaultCache  
*maxElementsInMemory*="1000"  
*eternal*="false"  
*timeToIdleSeconds*="120"  
*timeToLiveSeconds*="120"  
*overflowToDisk*="true"  
/>  
  
<cache *name*="Employee"  
 *maxElementsInMemory*="500"  
 *eternal*="true"  
 *timeToIdleSeconds*="0"  
 *timeToLiveSeconds*="0"  
 *overflowToDisk*="false"  
/>

Vậy là đủ, bây giờ chúng ta đã kích hoạt bộ nhớ cache cấp hai cho lớp Employee và Hibernate truy cập bộ nhớ cache cấp hai bất cứ khi nào bạn điều hướng tới Employee hoặc khi bạn tải Employee bằng mã nhận dạng.

Bạn nên phân tích tất cả các lớp và lựa chọn chiến lược bộ nhớ đệm thích hợp cho mỗi lớp. Đôi khi, bộ nhớ đệm cấp thứ hai có thể làm giảm hiệu suất của ứng dụng. Vì vậy, bạn nên chuẩn bị ứng dụng của mình trước mà không cần bật bộ nhớ đệm và sau đó cho phép bộ nhớ đệm phù hợp và kiểm tra hiệu suất. Nếu bộ nhớ đệm không cải thiện hiệu năng hệ thống thì không cần phải kích hoạt bất kỳ loại bộ nhớ đệm nào.

### Bộ nhớ cache truy vấn

Để sử dụng bộ nhớ cache truy vấn, trước tiên bạn phải kích hoạt nó bằng cách sử dụng thuộc tính **hibernate.cache.use\_query\_cache = "true"** trong tệp tin cấu hình. Bằng cách thiết lập thuộc tính này thành true, bạn cho phép Hibernate tạo ra các bộ đệm cần thiết trong bộ nhớ để giữ các tập truy vấn và bộ nhận dạng (identifier).

Tiếp theo, để sử dụng bộ nhớ truy vấn, bạn sử dụng phương thức setCacheable(Boolean) của lớp Query. Ví dụ:

Session session = SessionFactory.openSession();  
 Query query = session.createQuery("FROM EMPLOYEE");  
query.setCacheable(*true*);  
 *List* users = query.list();  
SessionFactory.closeSession();

Hibernate cũng hỗ trợ hỗ trợ bộ nhớ cache rất tinh vi thông qua khái niệm vùng nhớ cache. Một vùng nhớ cache là một phần của bộ nhớ cache được đặt tên.

Session session = SessionFactory.openSession();  
 Query query = session.createQuery("FROM EMPLOYEE");  
query.setCacheable(*true*);  
query.setCacheRegion("employee");  
 *List* users = query.list();  
SessionFactory.closeSession();

Đoạn code trên sử dụng phương thức để bảo Hibernate lưu trữ và tìm kiếm truy vấn trong vùng Employee của bộ nhớ cache.

## Hibernate - Batch Processing

Xem xét một tình huống khi bạn cần tải lên một số lượng lớn các bản ghi vào cơ sở dữ liệu của bạn bằng cách sử dụng Hibernate. Sau đây là đoạn code để đạt được điều này bằng cách sử dụng Hibernate:

Session session = SessionFactory.openSession();  
 Transaction tx = session.beginTransaction();  
*for* ( *int* i=0; i<100000; i++ ) {  
 Employee employee = *new* Employee(.....);  
 session.save(employee);  
 }  
tx.commit();  
session.close();

Mặc định, Hibernate sẽ lưu trữ tất cả các đối tượng đã tồn tại trong bộ nhớ cache cấp session và cuối cùng ứng dụng của bạn sẽ rơi vào tình trạng **OutOfMemoryException** khi ở hàng 50.000. Bạn có thể sử dụng **batch processing** trong Hibernate để giải quyết vấn đề này.

Để sử dụng tính năng batch processing trong Hibernate, đầu tiên hãy đặt **hibernate.jdbc.batch\_size** có kích thước batch là X = 10 đến 50 hoặc một số khác nào đó tùy thuộc vào kích thước đối tượng. Điều này sẽ nói với các Hibernate rằng mỗi lô có X bản ghi được insert theo batch. Để thực hiện điều này trong đoạn code trên, chúng ta sẽ cần phải sửa đổi như sau:

Session session = SessionFactory.openSession();  
 Transaction tx = session.beginTransaction();  
*for* ( *int* i=0; i<100000; i++ ) {  
 Employee employee = *new* Employee(.....);  
 session.save(employee);  
 *if*( i % 50 == 0 ) { *// Tương tự như kích thước batch JDBC  
 //flush một batch của các lệnh insert và giải phóng bộ nhớ* session.flush();  
 session.clear();  
 }  
 }  
tx.commit();  
session.close();

Đoạn code trên sẽ làm việc tốt cho các hoạt động INSERT, nhưng khi bạn thực hiện hoạt động UPDATE thì bạn có sử dụng code sau đây:

Session session = sessionFactory.openSession();  
 Transaction tx = session.beginTransaction();  
  
 *ScrollableResults* employeeCursor = session.createQuery("FROM EMPLOYEE").scroll();  
 *int* count = 0;  
  
*while* ( employeeCursor.next() ) {  
 Employee employee = (Employee) employeeCursor.get(0);  
 employee.updateEmployee();  
 seession.update(employee);  
 *if* ( ++count % 50 == 0 ) {  
 session.flush();  
 session.clear();  
 }  
 }  
tx.commit();  
session.close();

### Ví dụ về Batch Processing trong Hibernate

Hãy sửa file cấu hình để thêm tính năng **hibernate.jdbc.batch\_size:**

<?*xml version*='1.0' *encoding*='utf-8'?>  
<!DOCTYPE *hibernate-configuration* PUBLIC  
 "-//Hibernate/Hibernate Configuration DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  
  
<hibernate-configuration>  
 <session-factory>  
 <property *name*="connection.url">jdbc:mysql://localhost:3306/demo</property>  
 <property *name*="connection.driver\_class">com.mysql.cj.jdbc.Driver</property>  
 <property *name*="dialect">org.hibernate.dialect.MySQL5Dialect</property>  
 <property *name*="current\_session\_context\_class">thread</property>  
 <property *name*="hibernate.connection.username">root</property>  
 <property *name*="hibernate.connection.password">Admin123</property>  
 <property *name*="hibernate.hbm2ddl.auto">update</property>  
 <property *name*="hibernate.connection.autocommit">true</property>

//Chỉnh batch size tại đây:  
 <property *name*="hibernate.jdbc.batch\_size">50</property>  
 <property *name*="hibernate.show\_sql">true</property>  
 <mapping *class*="com.model.entity.TblAssistantEntity"/>  
 <mapping *class*="com.model.entity.TblClassEntity"/>  
 <mapping *class*="com.model.entity.TblStudentEntity"/>  
 <mapping *class*="com.model.entity.TblTeacherEntity"/>  
 </session-factory>  
  
  
</hibernate-configuration>

Lớp Student:

@Entity  
@Table(name = "tbl\_student", schema = "demo")  
*public class* TblStudentEntity {  
 @Id@GeneratedValue(strategy = GenerationType.IDENTITY)@Column(name = "id", nullable = *false*)  
 *private int* id;  
 @Basic@Column(name = "name", nullable = *false*, length = 255)  
 *private* String name;

//Constructor, getter setter etc…  
  
  
}

Lớp này tạo ra bảng:

*create table* tbl\_teacher  
(  
 id *int auto\_increment  
 primary key*,  
 *name varchar*(255) *not null*)

Với phương thức sau ta sẽ tạo ra 10000 student với name được đưa vào:

*public void* uc\_Create(String name) {  
 *Session* session = factory.getCurrentSession();  
 session.getTransaction().begin();  
 *for* ( *int* i=0; i<100000; i++ ) {  
 TblStudentEntity student = *new* TblStudentEntity();  
 student.setName(name);  
 session.save(student);  
 *if*( i % 50 == 0 ) {  
 session.flush();  
 session.clear();  
 }  
 session.getTransaction().commit();  
}

## Hibernate - Interceptors

Như bạn đã biết được rằng trong Hibernate, một đối tượng sẽ được tạo ra và tồn tại. Một khi đối tượng đã được thay đổi, nó phải được lưu trở lại cơ sở dữ liệu. Quá trình này tiếp tục cho đến khi đối tượng tiếp theo được yêu cầu, và nó sẽ được tải từ các persistent store.

Do đó, một đối tượng đi qua các giai đoạn khác nhau trong vòng đời của nó. Giao tiếp Interceptor trong Hibernate cung cấp các phương thức có thể được gọi ở các giai đoạn khác nhau để thực hiện một số nhiệm vụ được yêu cầu. Các phương thức này là gọi lại từ session tới ứng dụng, cho phép ứng dụng kiểm tra và/hoặc thao tác các thuộc tính của một đối tượng persistent trước khi nó được lưu, cập nhật, xóa hoặc nạp. Sau đây là danh sách tất cả các phương thức có sẵn trong Interceptor Interface:

|  |  |
| --- | --- |
| **No.** | **Phương thức và mô tả** |
| 1 | **findDirty()**  Phương thức này được gọi khi phương thức **flush() được gọi trên một đối tượng Session.** |
| 2 | **instantiate()**  Phương thức này được gọi khi một lớp persistent được khởi tạo. |
| 3 | **isUnsaved()**  Phương thức này được gọi khi một đối tượng được truyền vào phương thức **saveOrUpdate()**/ |
| 4 | **onDelete()**  Phương thức này được gọi trước khi một đối tượng bị xóa. |
| 5 | **onFlushDirty()**  Phương thức này được gọi khi Hibernate phát hiện ra rằng một đối tượng là dirty(tức là đã được thay đổi) trong quá trình flush nghĩa là thao tác update. |
| 6 | **onLoad()**  Phương thức này được gọi trước khi một đối tượng được khởi tạo. |
| 7 | **onSave()**  Phương thức này được gọi trước khi một đối tượng được lưu. |
| 8 | **postFlush()**  Phương thức này được gọi sau khi flush và một đối tượng đã được update trong bộ nhớ. |
| 9 | **preFlush()**  Phương thức này được gọi trước một flush. |

Hibernate Interceptor cho phép chúng ta kiểm soát toàn diện đối tượng trong cả ứng dụng và cơ sở dữ liệu.

### Ví dụ về Interceptor trong Hibernate

Để xây dựng một interceptor bạn có thể impliments lớp Interceptor trực tiếp hoặc extents lớp EmptyInterceptor. Sau đây là các bước đơn giản để sử dụng chức năng Interceptor trong Hibernate.

### Tạo các Interceptor

Tạo lớp MyInterceptor kế thừa EmptyInterceptor nơi phương thức của Interceptor sẽ được gọi tự động khi đối tượng Employee được tạo ra và cập nhật. Bạn có thể thực hiện nhiều phương thức hơn theo yêu cầu của bạn.

*public class* MyInterceptor *extends* EmptyInterceptor {  
  
 *public void* onDelete(Object entity, *Serializable* id, Object[] state, String[] propertyNames, Type[] types) {  
 *// do nothing* }  
  
 *// This method is called when Employee object gets updated.  
 public boolean* onFlushDirty(Object entity, *Serializable* id, Object[] currentState, Object[] previousState,  
 String[] propertyNames, Type[] types) {  
 *if* (entity *instanceof* Employee) {  
 System.out.println("Update Operation");  
 *return true*;  
 }  
 *return false*;  
 }  
  
 *public boolean* onLoad(Object entity, *Serializable* id, Object[] state, String[] propertyNames, Type[] types) {  
 *// do nothing  
 return true*;  
 }  
  
 *// This method is called when Employee object gets created.  
 public boolean* onSave(Object entity, *Serializable* id, Object[] state, String[] propertyNames, Type[] types) {  
 *if* (entity *instanceof* Employee) {  
 System.out.println("Create Operation");  
 *return true*;  
 }  
 *return false*;  
 }  
  
 *// called before commit into database  
 public void* preFlush(Iterator iterator) {  
 System.out.println("preFlush");  
 }  
  
 *// called after committed into database  
 public void* postFlush(Iterator iterator) {  
 System.out.println("postFlush");  
 }  
 }  
  
 }

### Tạo lớp POJO

File: Employee.java

*public class* Employee {  
 *private int* id;  
 *private* String firstName;  
 *private* String lastName;  
 *private int* salary;

//getter & setter & constructor  
  
}

### Tạo bảng cơ sở dữ liệu

Bước thứ hai, tạo các bảng trong cơ sở dữ liệu. Sẽ có một bảng tương ứng với mỗi đối tượng mà bạn sẵn sàng cung cấp persistent. Các đối tượng trên cần phải được lưu trữ và truy xuất vào bảng RDBMS sau:

*create table* EMPLOYEE (  
 id *INT NOT NULL auto\_increment*,  
 first\_name *VARCHAR*(20) *default NULL*,  
 last\_name *VARCHAR*(20) *default NULL*,  
 salary *INT default NULL*,  
 *PRIMARY KEY* (id)  
);

### Tạo file mapping

Bước này tạo ra một file ánh xạ hướng dẫn Hibernate làm thế nào để ánh xạ các lớp hoặc các lớp đã định nghĩa vào các bảng cơ sở dữ liệu.

File: Employee.hbm.xml

<?*xml version*="1.0" *encoding*="utf-8"?>  
<!DOCTYPE *hibernate-mapping* PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  
  
<hibernate-mapping>  
 <class *name*="hibernate.interceptor.entity.Employee" *table*="EMPLOYEE">  
 <meta *attribute*="class-description">  
 Lớp này chứa chi tiết về employee  
 </meta>  
 <id *name*="id" *type*="int" *column*="id">  
 <generator *class*="native"/>  
 </id>  
 <property *name*="firstName" *column*="first\_name" *type*="string"/>  
 <property *name*="lastName" *column*="last\_name" *type*="string"/>  
 <property *name*="salary" *column*="salary" *type*="int"/>  
 </class>  
</hibernate-mapping>

Tạo file cấu hình Hibernate

File: hibernate.cfg.xml

<?*xml version*="1.0" *encoding*="UTF-8"?>  
<!DOCTYPE *hibernate-configuration* SYSTEM  
 "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  
  
<hibernate-configuration>  
 <session-factory>  
 <property *name*="hibernate.dialect">  
 org.hibernate.dialect.MySQLDialect  
 </property>  
 <property *name*="hibernate.connection.driver\_class">  
 com.mysql.jdbc.Driver  
 </property>  
  
 *<!-- Thông tin kết nối db -->* <property *name*="hibernate.connection.url">  
 jdbc:mysql://localhost/demo  
 </property>  
 <property *name*="hibernate.connection.username">  
 root  
 </property>  
 <property *name*="hibernate.connection.password">  
 Admin123  
 </property>  
  
 *<!-- List các file XML mapping -->* <mapping *resource*="Employee.hbm.xml" />  
  
 </session-factory>  
</hibernate-configuration>

Tạo lớp ứng dụng

Cuối cùng, chúng ta sẽ tạo lớp chứa phương thức main() để chạy ứng dụng. Ở đây cần lưu ý rằng trong khi tạo đối tượng Session, chúng ta đã sử dụng lớp Interceptor như một đối số.

*import* org.hibernate.HibernateException;  
  
*public class* ManageEmployee {  
 *private static* SessionFactory *factory*;  
  
 *public static void* main(String[] args) {  
 *try* {  
 *factory* = *new* Configuration().configure().buildSessionFactory();  
 } *catch* (Throwable ex) {  
 System.err.println("Failed to create sessionFactory object." + ex);  
 *throw new* ExceptionInInitializerError(ex);  
 }  
  
 ManageEmployee ME = *new* ManageEmployee();  
  
 *// Add few employee records in database* Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);  
 Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);  
 Integer empID3 = ME.addEmployee("John", "Paul", 10000);  
  
 *// List down all the employees* ME.listEmployees();  
  
 *// Update employee's records* ME.updateEmployee(empID1, 5000);  
  
 *// Delete an employee from the database* ME.deleteEmployee(empID2);  
  
 *// List down new list of the employees* ME.listEmployees();  
 }  
  
 *// Method to CREATE an employee in the database  
 public* Integer addEmployee(String fname, String lname, *int* salary) {  
 Session session = *factory*.openSession(*new* MyInterceptor());  
 Transaction tx = *null*;  
 Integer employeeID = *null*;  
 *try* {  
 tx = session.beginTransaction();  
 Employee employee = *new* Employee(fname, lname, salary);  
 employeeID = (Integer) session.save(employee);  
 tx.commit();  
 } *catch* (HibernateException e) {  
 *if* (tx != *null*)  
 tx.rollback();  
 e.printStackTrace();  
 } *finally* {  
 session.close();  
 }  
 *return* employeeID;  
 }  
  
 *// Method to READ all the employees  
 public void* listEmployees() {  
 Session session = *factory*.openSession(*new* MyInterceptor());  
 Transaction tx = *null*;  
 *try* {  
 tx = session.beginTransaction();  
 List employees = session.createQuery("FROM Employee").list();  
 *for* (Iterator iterator = employees.iterator(); iterator.hasNext();) {  
 Employee employee = (Employee) iterator.next();  
 System.out.print("First Name: " + employee.getFirstName());  
 System.out.print(" Last Name: " + employee.getLastName());  
 System.out.println(" Salary: " + employee.getSalary());  
 }  
 tx.commit();  
 } *catch* (HibernateException e) {  
 *if* (tx != *null*)  
 tx.rollback();  
 e.printStackTrace();  
 } *finally* {  
 session.close();  
 }  
 }  
  
 *// Method to UPDATE salary for an employee  
 public void* updateEmployee(Integer EmployeeID, *int* salary) {  
 Session session = *factory*.openSession(*new* MyInterceptor());  
 Transaction tx = *null*;  
 *try* {  
 tx = session.beginTransaction();  
 Employee employee = (Employee) session.get(Employee.*class*, EmployeeID);  
 employee.setSalary(salary);  
 session.update(employee);  
 tx.commit();  
 } *catch* (HibernateException e) {  
 *if* (tx != *null*)  
 tx.rollback();  
 e.printStackTrace();  
 } *finally* {  
 session.close();  
 }  
 }  
  
 *// Method to DELETE an employee from the records  
 public void* deleteEmployee(Integer EmployeeID) {  
 Session session = *factory*.openSession(*new* MyInterceptor());  
 Transaction tx = *null*;  
 *try* {  
 tx = session.beginTransaction();  
 Employee employee = (Employee) session.get(Employee.*class*, EmployeeID);  
 session.delete(employee);  
 tx.commit();  
 } *catch* (HibernateException e) {  
 *if* (tx != *null*)  
 tx.rollback();  
 e.printStackTrace();  
 } *finally* {  
 session.close();  
 }  
 }  
}

Kết quả run trên Eclipse

Create Operation

preFlush

postFlush

Create Operation

preFlush

postFlush

Create Operation

preFlush

postFlush

First Name: Zara Last Name: Ali Salary: 1000

First Name: Daisy Last Name: Das Salary: 5000

First Name: John Last Name: Paul Salary: 10000

preFlush

postFlush

preFlush

Update Operation

postFlush

preFlush

postFlush

First Name: Zara Last Name: Ali Salary: 5000

First Name: John Last Name: Paul Salary: 10000

preFlush

postFlush

### Tham khảo:

Tutorialpont, Viettut…