[https://docs.jboss.org/hibernate/orm/current/userguide/html\_single/Hibernate\_User\_Guide.html#basic-nationalized](https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html" \l "basic-nationalized)

[Hibernate 5.3.15 1](#_Toc34122162)

[Kiến trúc Hibernate 1](#_Toc34122163)

[Tổng quan về Hibernate 1](#_Toc34122164)

[Mô hình miền - Domain Model 4](#_Toc34122165)

[2.1. Ánh xạ kiểu (Mapping types ) 4](#_Toc34122166)

[2.2. Chiến lược đặt tên - Naming strategies 6](#_Toc34122167)

[2.2.1. ImplicitNamingStrategy 6](#_Toc34122168)

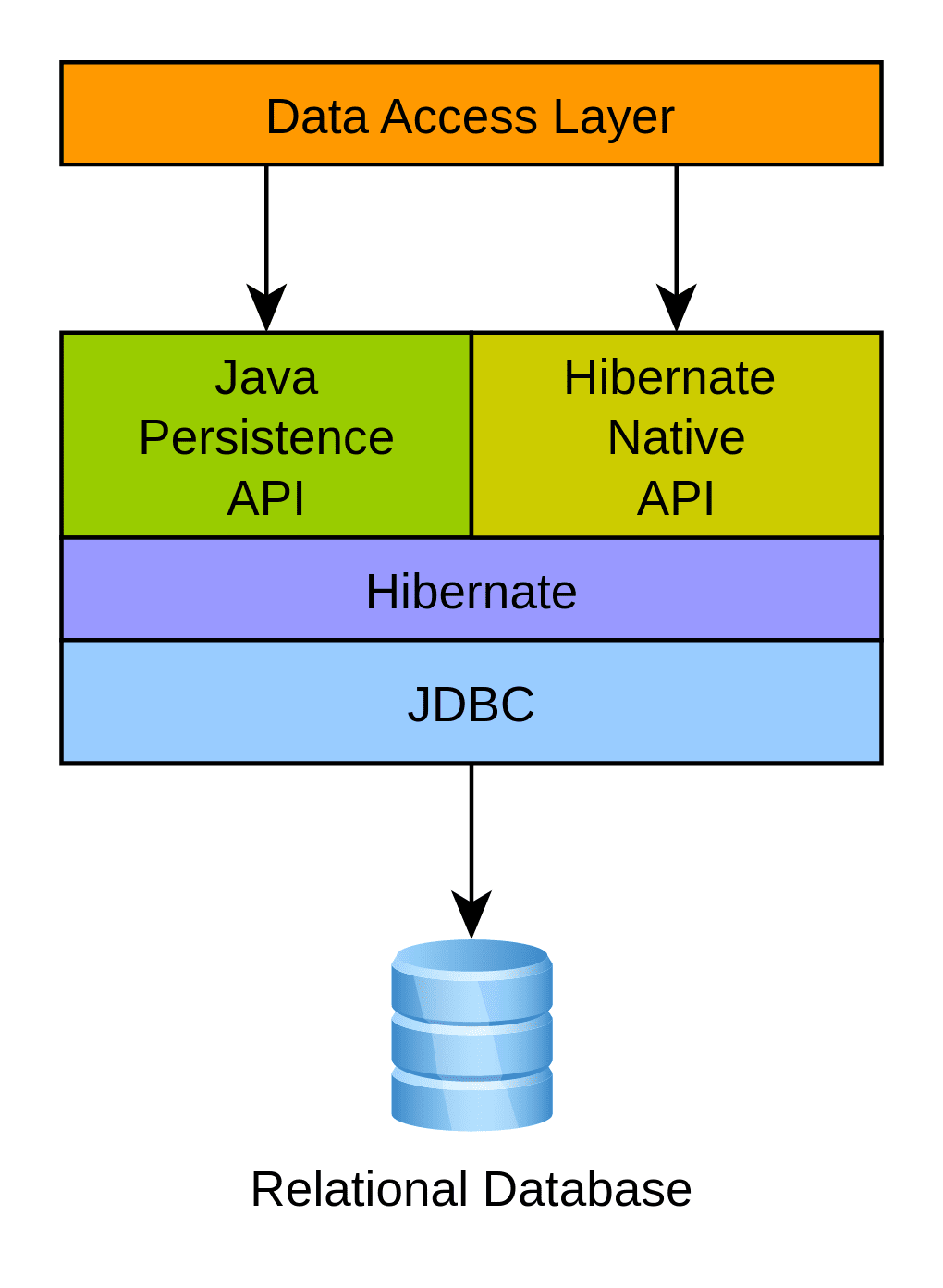
[2.3. Các kiểu cơ bản - Basic Types 9](#_Toc34122169)

# Hibernate 5.3.15

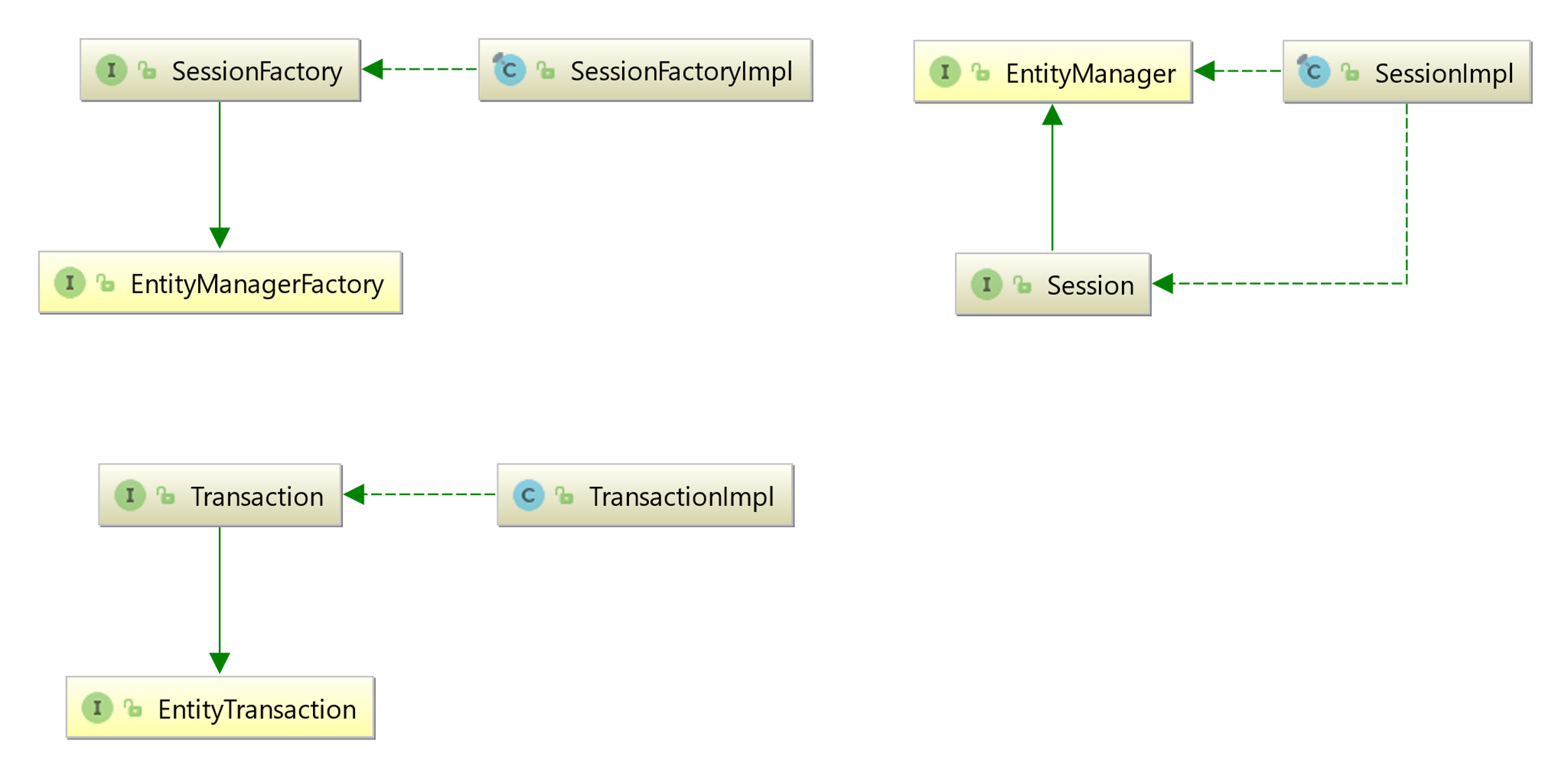
## Kiến trúc Hibernate

### Tổng quan về Hibernate

Hibernate, là một giải pháp ORM, "nằm giữa" lớp truy cập dữ liệu ứng dụng Java (Java application data Access) và cơ sở dữ liệu quan hệ (Relational Database), như có thể thấy trong sơ đồ, ứng dụng Java sử dụng API Hibernate để tải, lưu trữ, truy vấn, v.v. dữ liệu miền của nó.



Là một JPA provider, Hibernate triển khai những đặc điểm của Java Persistence API. Liên kết giữa JPA interfaces và các triển khai cụ thể của Hibernate được trình bày trong biểu đồ sau:



SessionFactory (org.hibernate.SessionFactory)

Đại diện cho việc ánh xạ giữa mô hình miền ứng dụng (application domain model) và cơ sở dữ liệu (database). Nó là an toàn với luồng và bất biến. Có khả năng tạo ra những thể hiện của org.hibernate.Session .

EntityManagerFactory  tương đương với JPA (Java Persistent API) của SessionFactory. Về căn bản, cả hai cùng đưa đến một triển khai của SessionFactory.

Một SessionFactory là một đối tượng nặng khi tạo. Vậy nên, với mỗi cơ sở dữ liệu, chương trình chỉ nên có duy nhất một liên kết SessionFactory. SessionFactory duy trì những dịch vụ mà Hibernate sử dụng trên tất cả những Session như là cache cấp 2, connection pools, transaction system integrations…

Session (org.hibernate.Session)

Là một đối tượng đơn luồng, ngắn hạn mô hình hóa khái niệm “Đơn vị công việc” (Unit of Work) [PoEAA](https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#PoEAA). Trong danh pháp JPA,  Session được đại diện bởi một EntityManager.

Phía trong nó, Session chứa một JDBC java.sql.Connection và có khả năng tạo ra những thể hiện của  org.hibernate.Transaction .

Nó duy trì những persistence context có khả năng đọc lại (repeatable read -cache level 1) của mô hình miền ứng dụng (application domain model).

Transaction (org.hibernate.Transaction)

Là một dối tượng đơn luồng, ngắn hạn được ứng dụng sử dụng để vạch ra ranh giới giữa các giao dịch vật lý riêng biệt. EntityTransaction  tương đương với JPA và cả hai hoạt động như một API trừu tượng để phân tách ứng dụng khỏi hệ thống transaction nền tảng đang sử dụng (JDBC hoặc JTA). .

## Mô hình miền - Domain Model

Thuật ngữ [domain model](https://en.wikipedia.org/wiki/Domain_model)  xuất phát từ lĩnh vực mô hình dữ liệu. Đây là mô hình mô tả về miền vấn đề bạn đang làm việc. Đôi khi, bạn cũng sẽ nghe thấy thuật ngữ *persistent classes*.

Mô hình miền ứng dụng là đặc tính trung tâm của ORM. Chúng tạo nên các lớp mà bạn muốn liên kết. Hibernate hoạt động tốt nhất nếu các lớp này tuân theo mô hình lập trình Plain Old Object (POJO) / JavaBean.

Trước đây, các ứng dụng sử dụng Hibernate sẽ sử dụng định dạng tệp ánh xạ XML riêng của nó cho mục đích này. Tuy nhiên đã lỗi thời, và được thay thế bằng cách sử dụng Annotation.

### 2.1. Ánh xạ kiểu (Mapping types )

Hibernate có thể hiểu cả biểu diễn Java lẫn JDBC của dữ liệu ứng dụng. Khả năng đọc/ghi dữ liệu này từ/đến cơ sở dữ liệu là chức năng của một kiểu Hibernate. Một kiểu, trong trường hợp này, là một triển khai của giao org.hibernate.type.Type. Kiểu trong Hibernate cũng mô tả các khía cạnh hành vi khác nhau của loại Java như cách kiểm tra sự bằng nhau, sao chép các giá trị…

Ví dụ 1. Một bảng đơn giản và một domain model.

*create table* Contact (  
 id *integer not null*,  
 *first varchar*(255),  
 *last varchar*(255),  
 middle *varchar*(255),  
 notes *varchar*(255),  
 starred *boolean not null*,  
 website *varchar*(255),  
 *primary key* (id)  
)

Domain Model:

@Entity(name = "Contact")  
*public static class* Contact {  
  
 @Id  
 *private* Integer id;  
  
 *private* Name name;  
  
 *private* String notes;  
  
 *private* URL website;  
  
 *private boolean* starred;  
  
 *//Getters and setters are omitted for brevity*}  
  
@Embeddable  
*public class* Name {  
  
 *private* String first;  
  
 *private* String middle;  
  
 *private* String last;  
  
 *// getters and setters omitted*}

Hibernate phân các types thành 2 nhóm:

* [Value types](https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#categorization-value)
* [Entity types](https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#categorization-entity)

#### 2.1.1. Kiểu giá trị - Value types

Một Value types là một phần dữ liệu không xác định vòng đời của chính nó. Trên thực tế, nó thuộc sở hữu của một thực thể, thứ xác định vòng đời của nó.

Nhìn theo một cách khác, tất cả trạng thái của một thực thể (entity) được tạo thành hoàn toàn từ các loại giá trị. Các trường trạng thái hoặc thuộc tính JavaBean được gọi là các *persistent attributes*. Persistent attributes của lớp Contact là các Value types.

Các Value types được chia làm ba loại:

Kiểu cơ bản:

Bên trong bảng Contact, tất cả các thuộc tính ngoại trừ name là kiểu cơ bản.

Kiểu nhúng (Embeddable types)

Thuộc tính name trong bảng chính là một kiểu nhúng.

Kiểu Collection (Collection types)

Dù không được đề cập trong ví dụ, nhưng chúng ta còn có một kiểu dữ liệu Collection.

#### 2.1.2. Kiểu thực thể - Entity types

Kiểu thực thể tồn tại độc lập với các đối tượng khác trong khi các kiểu giá trị thì không. Các thực thể là các lớp mô hình miền tương quan với các hàng (row) trong các bảng cơ sở dữ liệu (database), sử dụng một mã định danh duy nhất (unique identifier). Do một thực thể chỉ có một mã định danh duy nhất, nên các thực thể tồn tại độc lập và xác định vòng đời của chính chúng. Bản thân lớp Contact là một ví dụ về một thực thể.

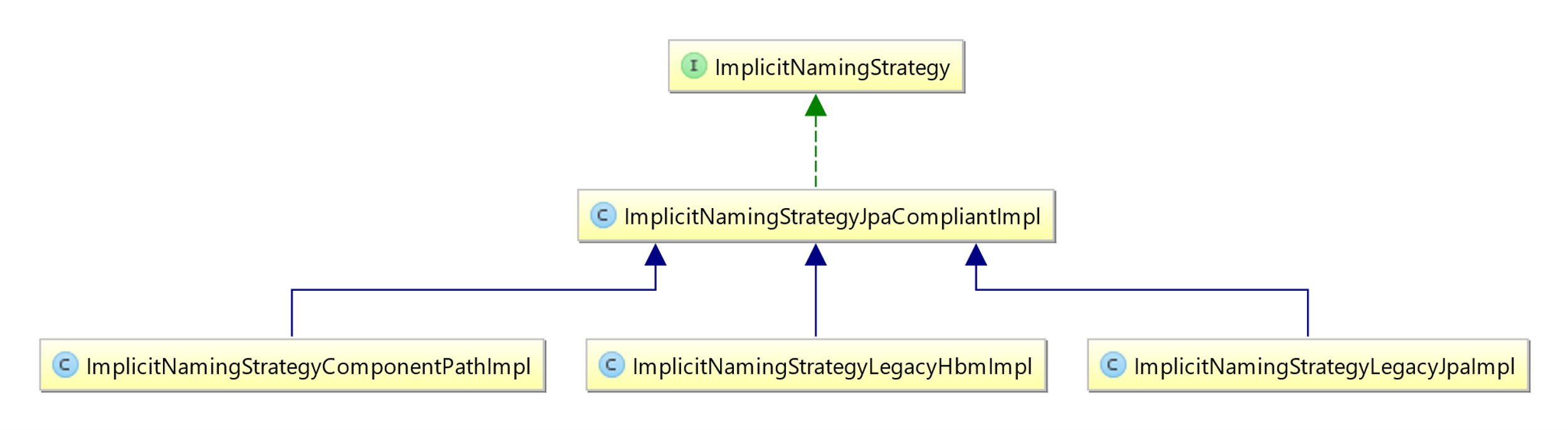
### 2.2. Chiến lược đặt tên - Naming strategies

Ánh xạ một Đối tượng vào cơ sở dữ liệu quan hệ tương ứng được thông qua 2 bước:

* Đầu tiên xác định một cái tên logic trong sơ đồ miền. Nó có thể được chỉ định rõ ràng bởi người dùng (thông qua @Column , @Table…) hoặc là Hibernate ngầm định. Thông qua [ImplicitNamingStrategy](https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#ImplicitNamingStrategy) contract.
* Tiếp theo là việc phân giải tên logic này thành tên vật lý được xác định bởi [PhysicalNamingStrategy](https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#PhysicalNamingStrategy).

### 2.2.1. ImplicitNamingStrategy

Khi một thực thể không được định rõ bảng mà nó ánh xạ tới, lúc này org.hibernate.boot.model.naming.ImplicitNamingStrategy sẽ ngầm định xác định tên logic ánh xạ của nó giống với thuộc tính và tên class.



#### 2.2.2. PhysicalNamingStrategy

Nhiều tổ chức định nghĩa các quy tắc xung quanh việc đặt tên của các đối tượng cơ sở dữ liệu (bảng, cột, khóa ngoại, v.v.). Ý tưởng của PhysNamingStrargety là giúp thực hiện các quy tắc đặt tên như vậy mà không cần phải mã hóa chúng vào ánh xạ thông qua các tên rõ ràng.

Khi không được chỉ định rõ ràng, ImplicitNamingStrargety xác định rằng một thuộc tính có tên là accountNumber ánh xạ tới một tên cột logic accountNumber. Tuy nhiên với, PhysNamingStrargety, ta có thể chỉ định cột vật lý nên được viết tắt là acct\_num .

Để làm được điều này, ta có thể xây dựng chiến lược riêng cho việc đặt tên.Ví dụ như:

*/\*  
 \* Hibernate, Relational Persistence for Idiomatic Java  
 \*  
 \* License: GNU Lesser General Public License (LGPL), version 2.1 or later.  
 \* See the lgpl.txt file in the root directory or <http://www.gnu.org/licenses/lgpl-2.1.html>.  
 \*/  
package* org.hibernate.userguide.naming;  
  
*import* java.util.LinkedList;  
*import* java.util.List;  
*import* java.util.Locale;  
*import* java.util.Map;  
*import* java.util.TreeMap;  
  
*import* org.hibernate.boot.model.naming.Identifier;  
*import* org.hibernate.boot.model.naming.*PhysicalNamingStrategy*;  
*import* org.hibernate.engine.jdbc.env.spi.*JdbcEnvironment*;  
  
*import* org.apache.commons.lang3.StringUtils;  
  
*/\*\*  
 \* An example PhysicalNamingStrategy that implements database object naming standards  
 \* for our fictitious company Acme Corp.  
 \* <p/>  
 \* In general Acme Corp prefers underscore-delimited words rather than camel casing.  
 \* <p/>  
 \* Additionally standards call for the replacement of certain words with abbreviations.  
 \*  
 \** ***@author*** *Steve Ebersole  
 \*/  
public class* AcmeCorpPhysicalNamingStrategy *implements PhysicalNamingStrategy* {  
 *private static final* Map<String,String> ABBREVIATIONS = *buildAbbreviationMap*();  
  
 @Override  
 *public* Identifier toPhysicalCatalogName(Identifier name, *JdbcEnvironment* jdbcEnvironment) {  
 *// Acme naming standards do not apply to catalog names  
 return* name;  
 }  
  
 @Override  
 *public* Identifier toPhysicalSchemaName(Identifier name, *JdbcEnvironment* jdbcEnvironment) {  
 *// Acme naming standards do not apply to schema names  
 return* name;  
 }  
  
 @Override  
 *public* Identifier toPhysicalTableName(Identifier name, *JdbcEnvironment* jdbcEnvironment) {  
 *final* List<String> parts = splitAndReplace( name.getText() );  
 *return* jdbcEnvironment.getIdentifierHelper().toIdentifier(  
 join( parts ),  
 name.isQuoted()  
 );  
 }  
  
 @Override  
 *public* Identifier toPhysicalSequenceName(Identifier name, *JdbcEnvironment* jdbcEnvironment) {  
 *final* LinkedList<String> parts = splitAndReplace( name.getText() );  
 *// Acme Corp says all sequences should end with \_seq  
 if* ( !"seq".equalsIgnoreCase( parts.getLast() ) ) {  
 parts.add( "seq" );  
 }  
 *return* jdbcEnvironment.getIdentifierHelper().toIdentifier(  
 join( parts ),  
 name.isQuoted()  
 );  
 }  
  
 @Override  
 *public* Identifier toPhysicalColumnName(Identifier name, *JdbcEnvironment* jdbcEnvironment) {  
 *final* List<String> parts = splitAndReplace( name.getText() );  
 *return* jdbcEnvironment.getIdentifierHelper().toIdentifier(  
 join( parts ),  
 name.isQuoted()  
 );  
 }  
  
 *private static* Map<String, String> buildAbbreviationMap() {  
 TreeMap<String,String> abbreviationMap = *new* TreeMap<> ( String.CASE\_INSENSITIVE\_ORDER );  
 abbreviationMap.put( "account", "acct" );  
 abbreviationMap.put( "number", "num" );  
 *return* abbreviationMap;  
 }  
  
 *private* LinkedList<String> splitAndReplace(String name) {  
 LinkedList<String> result = *new* LinkedList<>();  
 *for* ( String part : StringUtils.splitByCharacterTypeCamelCase( name ) ) {  
 *if* ( part == *null* || part.trim().isEmpty() ) {  
 *// skip null and space  
 continue*;  
 }  
 part = applyAbbreviationReplacement( part );  
 result.add( part.toLowerCase( Locale.ROOT ) );  
 }  
 *return* result;  
 }  
  
 *private* String applyAbbreviationReplacement(String word) {  
 *if* ( ABBREVIATIONS.containsKey( word ) ) {  
 *return* ABBREVIATIONS.get( word );  
 }  
  
 *return* word;  
 }  
  
 *private* String join(List<String> parts) {  
 *boolean* firstPass = *true*;  
 String separator = "";  
 StringBuilder joined = *new* StringBuilder();  
 *for* ( String part : parts ) {  
 joined.append( separator ).append( part );  
 *if* ( firstPass ) {  
 firstPass = *false*;  
 separator = "\_";  
 }  
 }  
 *return* joined.toString();  
 }  
}

### 2.3. Các kiểu cơ bản - Basic Types

Các basic type thường ánh xạ một cột cơ sở dữ liệu tới một kiểu Java đơn lẻ. Hibernate cung cấp một số basic type tích hợp sẵn, tuân theo các ánh xạ tự nhiên được đề xuất bởi JDBC.

#### 2.3.1. Kiểu cơ bản mà Hibernate cung cấp sẵn - Hibernate-provided BasicTypes

| **Hibernate type (org.hibernate.type package)** | **JDBC type** | **Java type** | **BasicTypeRegistry key(s)** |
| --- | --- | --- | --- |
| StringType | VARCHAR | java.lang.String | string, java.lang.String |
| MaterializedClob | CLOB | java.lang.String | materialized\_clob |
| TextType | LONGVARCHAR | java.lang.String | text |
| CharacterType | CHAR | char, java.lang.Character | char, java.lang.Character |
| BooleanType | BIT | boolean, java.lang.Boolean | boolean, java.lang.Boolean |
| NumericBooleanType | INTEGER, 0 is false, 1 is true | boolean, java.lang.Boolean | numeric\_boolean |
| YesNoType | CHAR, 'N'/'n' is false, 'Y'/'y' is true. The uppercase value is written to the database. | boolean, java.lang.Boolean | yes\_no |
| TrueFalseType | CHAR, 'F'/'f' is false, 'T'/'t' is true. The uppercase value is written to the database. | boolean, java.lang.Boolean | true\_false |
| ByteType | TINYINT | byte, java.lang.Byte | byte, java.lang.Byte |
| ShortType | SMALLINT | short, java.lang.Short | short, java.lang.Short |
| IntegerTypes | INTEGER | int, java.lang.Integer | int, java.lang.Integer |
| LongType | BIGINT | long, java.lang.Long | long, java.lang.Long |
| FloatType | FLOAT | float, java.lang.Float | float, java.lang.Float |
| DoubleType | DOUBLE | double, java.lang.Double | double, java.lang.Double |
| BigIntegerType | NUMERIC | java.math.BigInteger | big\_integer, java.math.BigInteger |
| BigDecimalType | NUMERIC | java.math.BigDecimal | big\_decimal, java.math.bigDecimal |
| TimestampType | TIMESTAMP | java.sql.Timestamp | timestamp, java.sql.Timestamp |
| TimeType | TIME | java.sql.Time | time, java.sql.Time |
| DateType | DATE | java.sql.Date | date, java.sql.Date |
| CalendarType | TIMESTAMP | java.util.Calendar | calendar, java.util.Calendar |
| CalendarDateType | DATE | java.util.Calendar | calendar\_date |
| CalendarTimeType | TIME | java.util.Calendar | calendar\_time |
| CurrencyType | VARCHAR | java.util.Currency | currency, java.util.Currency |
| LocaleType | VARCHAR | java.util.Locale | locale, java.utility.locale |
| TimeZoneType | VARCHAR, using the TimeZone ID | java.util.TimeZone | timezone, java.util.TimeZone |
| UrlType | VARCHAR | java.net.URL | url, java.net.URL |
| ClassType | VARCHAR (class FQN) | java.lang.Class | class, java.lang.Class |
| BlobType | BLOB | java.sql.Blob | blob, java.sql.Blob |
| ClobType | CLOB | java.sql.Clob | clob, java.sql.Clob |
| BinaryType | VARBINARY | byte[] | binary, byte[] |
| MaterializedBlobType | BLOB | byte[] | materialized\_blob |
| ImageType | LONGVARBINARY | byte[] | image |
| WrapperBinaryType | VARBINARY | java.lang.Byte[] | wrapper-binary, Byte[], java.lang.Byte[] |
| CharArrayType | VARCHAR | char[] | characters, char[] |
| CharacterArrayType | VARCHAR | java.lang.Character[] | wrapper-characters, Character[], java.lang.Character[] |
| UUIDBinaryType | BINARY | java.util.UUID | uuid-binary, java.util.UUID |
| UUIDCharType | CHAR, can also read VARCHAR | java.util.UUID | uuid-char |
| PostgresUUIDType | PostgreSQL UUID, through Types#OTHER, which complies to the PostgreSQL JDBC driver definition | java.util.UUID | pg-uuid |
| SerializableType | VARBINARY | implementors of java.lang.Serializable | Unlike the other value types, multiple instances of this type are registered. It is registered once under java.io.Serializable, and registered under the specific java.io.Serializable implementation class names. |
| StringNVarcharType | NVARCHAR | java.lang.String | nstring |
| NTextType | LONGNVARCHAR | java.lang.String | ntext |
| NClobType | NCLOB | java.sql.NClob | nclob, java.sql.NClob |
| MaterializedNClobType | NCLOB | java.lang.String | materialized\_nclob |
| PrimitiveCharacterArrayNClobType | NCHAR | char[] | N/A |
| CharacterNCharType | NCHAR | java.lang.Character | ncharacter |
| CharacterArrayNClobType | NCLOB | java.lang.Character[] | N/A |

| *Table 2. Java 8 BasicTypes* | | | |
| --- | --- | --- | --- |
| **Hibernate type (org.hibernate.type package)** | **JDBC type** | **Java type** | **BasicTypeRegistry key(s)** | |
| DurationType | BIGINT | java.time.Duration | Duration, java.time.Duration | |
| InstantType | TIMESTAMP | java.time.Instant | Instant, java.time.Instant | |
| LocalDateTimeType | TIMESTAMP | java.time.LocalDateTime | LocalDateTime, java.time.LocalDateTime | |
| LocalDateType | DATE | java.time.LocalDate | LocalDate, java.time.LocalDate | |
| LocalTimeType | TIME | java.time.LocalTime | LocalTime, java.time.LocalTime | |
| OffsetDateTimeType | TIMESTAMP | java.time.OffsetDateTime | OffsetDateTime, java.time.OffsetDateTime | |
| OffsetTimeType | TIME | java.time.OffsetTime | OffsetTime, java.time.OffsetTime | |
| ZonedDateTimeType | TIMESTAMP | java.time.ZonedDateTime | ZonedDateTime, java.time.ZonedDateTime | |

| *Table 3. Hibernate Spatial BasicTypes* | | | |
| --- | --- | --- | --- |
| **Hibernate type (org.hibernate.spatial package)** | **JDBC type** | **Java type** | **BasicTypeRegistry key(s)** | |
| JTSGeometryType | depends on the dialect | com.vividsolutions.jts.geom.Geometry | jts\_geometry, or the class name of Geometry or any of its subclasses | |
| GeolatteGeometryType | depends on the dialect | org.geolatte.geom.Geometry | geolatte\_geometry, or the class name of Geometry or any of its subclasses | |

#### 2.3.2. Annotation @Basic

Một kiểu cơ bản được chú thích bằng annotation javax.persistence.Basic . @Basic là một annotation mặc định, nên có thể được bỏ qua.

@Basic xác định hai thuộc tính:

optional - boolean (defaults to true)

Mặc định rằng thuộc tính này cho phép null. Trừ phi bản thân cột có kiểu dữ liệu nguyên thủy, còn không thì nó sẽ là NULLABLE.

fetch - FetchType (defaults to EAGER)

Xác định xem thuộc tính này nên được nạp một cách “eagerly” hay “lazily”.

#### 2.3.3. The @Column annotation

JPA có những quy tắc ngầm định để đặt tên bảng và cột.

Với những kiểu thuộc tính căn bản, quy tắc đặt tên ngầm định là tên cột giống với tên thuộc tính. Hoặc là bạn có thể chỉ rõ tên cột thay vì sử dụng mặc định.

*Example 5. Explicit column naming*

@Entity(name = "Product")  
*public class* Product {  
  
 @Id  
 *private* Integer id;  
  
 *private* String sku;  
  
 *private* String name;  
  
 @Column( name = "NOTES" )  
 *private* String description;  
}

Ví dụ như tại đây, @Column ánh xạ NOTE với thuộc tính description

#### 2.3.4. BasicTypeRegistry

Ta đã biết rằng kiểu trong Hibernate khác với kiểu trong Java lẫn SQL. Tuy nhiên Hibernate đã có khả năng ánh xạ chúng với nhau. Điển hình như org.hibernate.type.StringType ánh xạ với thuộc tính java.lang.String, hay  org.hibernate.type.IntegerType ánh xạ với  java.lang.Integer .

Điều này có được bởi một dịch vụ bên trong Hibernate gọi là org.hibernate.type.BasicTypeRegistry, về cơ bản nó cung cấp một collection map org.hibernate.type.BasicType với các khóa và tên.

Như ở phần trên, thuộc tính String sku trong bảng Product do không được chỉ định kiểu ánh xạ rõ ràng, nên Hibernate sẽ tự thông qua BasicTypeRegistry để tìm kiểu ánh xạ cho java.lang.String, kết quả Hibernate sẽ ưu tiên String thành Varchar.

#### 2.3.5. Explicit BasicTypes

Đôi lúc ta muốn một thuộc tính cụ thể được xử lý khác với mặc định.

Trong trường hợp này, ta cần phải chỉ rõ cho Hibernate BasicType nào cần sử dụng, thông qua org.hibernate.annotations.Type annotation.

@Entity(name = "Product")  
*public class* Product {  
  
 @Id  
 *private* Integer id;  
  
 *private* String sku;  
  
 @org.hibernate.annotations.Type( type = "nstring" )  
 *private* String name;  
  
 @org.hibernate.annotations.Type( type = "materialized\_nclob" )  
 *private* St

#### 2.3.6. Custom BasicTypes

Hibernate giúp các nhà phát triển tạo ra loại ánh xạ loại cơ bản của riêng họ tương đối dễ dàng. Ví dụ: bạn có thể muốn duy trì các thuộc tính của loại java.util.BigInteger  cho các cột VARCHAR hoặc hỗ trợ các loại hoàn toàn mới.

Có hai cách tiếp cận để phát triển một loại tùy chỉnh:

Triển khai BasicType và đăng ký nó.

Triển khai UserType mà không yêu cầu đăng ký kiểu.

#### 2.3.7. Mapping enums

Hibernate hỗ trợ ánh xạ Java enums như một giá trị cơ bản theo một vài cách.

##### **@Enumerated**

JPA ban đầu tuân thủ theo phương pháp map enums thông @Enumerated hoặc @MapKeyEnumerated cho map key, làm việc theo nguyên tắc rằng các giá trị enum được lưu trữ theo một trong 2 chiến lược được chỉ định bởi javax.persistence.EnumType:

ORDINAL

Lưu trữ giá trị của enum theo thứ tự tự nhiên, được chỉ định bởi java.lang.Enum#ordinal

STRING

Lưu trữ theo tên của giá trị enum, chỉ định bởi java.lang.Enum#name

@Enumerated(ORDINAL)*ví dụ:*

@Entity(name = "Phone")  
*public static class* Phone {  
  
 @Id  
 *private* Long id;  
  
 @Column(name = "phone\_number")  
 *private* String number;  
  
 @Enumerated(EnumType.ORDINAL)  
 @Column(name = "phone\_type")  
 *private* PhoneType type;  
  
 *//Getters and setters are omitted for brevity*}

Với code sau:

Phone phone = *new* Phone( );  
 phone.setId( 1L );  
 phone.setNumber( "123-456-78990" );  
 phone.setType( PhoneType.MOBILE );  
 entityManager.persist( phone );

SQL sản sinh:

*INSERT INTO* Phone (phone\_number, phone\_type, id)  
*VALUES* ('123-456-78990', 2, 1)

@Enumerated(STRING)*ví dụ:*

@Entity(name = "Phone")  
*public static class* Phone {  
  
 @Id  
 *private* Long id;  
  
 @Column(name = "phone\_number")  
 *private* String number;  
  
 @Enumerated(EnumType.STRING)  
 @Column(name = "phone\_type")  
 *private* PhoneType type;  
  
 *//Getters and setters are omitted for brevity*}

Cùng với code trên, lần này ta lại được mã SQL:

*INSERT INTO* Phone (phone\_number, phone\_type, id)  
*VALUES* ('123-456-78990', 'MOBILE', 1)

*Enum mapping with*AttributeConverter

##### **AttributeConverter**

Ở phần này, ta sẽ thay Female bằng F, và Male bằng M với Annotation @Converter

@Entity(name = "Person")  
*public static class* Person {  
 @Id  
 *private* Long id;  
 *private* String name;  
 @Convert( converter = GenderConverter.*class* )  
 *public* Gender gender;  
 *//Getters and setters are omitted for brevity*}  
@Converter  
*public static class* GenderConverter  
 *implements* AttributeConverter<Gender, Character> {  
 *public* Character convertToDatabaseColumn( Gender value ) {  
 *if* ( value == *null* ) {  
 *return null*;  
 }  
 *return* value.getCode();  
 }  
 *public* Gender convertToEntityAttribute( Character value ) {  
 *if* ( value == *null* ) {  
 *return null*;  
 }  
 *return* Gender.fromCode( value );  
 }  
}

#### 2.3.8. Mapping LOBs

Mapping LOBs (database Large Objects) Có hai dạng, dạng sử dụng các kiểu JDBC locator và dạng cụ thể hóa dữ liệu LOB.

Các JDBC LOB locators tồn tại để truy cập dữ tới dữ liệu LOB data một cách hiệu quả. Chúng cho phép JDBC driver stream từng phần của LOB data khi cần, có khả năng giải phòng bộ nhớ. Tuy nhiên, chúng cũng có những hạn chế nhất định. Ví dụ như, một LOB locator chỉ có hiệu lực trong thời gian transaction được thực hiện.

Ý tưởng vật chất hóa LOB là đánh đổi hiệu năng (không phải tất cả các driver đều xử lý dữ liệu LOB hiệu quả) cho mô hình lập trình tự nhiên hơn bằng cách sử dụng các loại Java quen thuộc như String hoặc byte [], v.v. cho các LOB này.

Vật chất hóa phân phối toàn bộ nội dung LOB trong bộ nhớ, trong khi đó, LOB locator (về lý thuyết) cho phép truyền các phần của nội dung LOB vào bộ nhớ khi cần.

Cách mapping dữ liệu CLOB chính là sử dụng annotation @Lob.

@Entity(name = "Product")  
*public static class* Product {  
  
 @Id  
 *private* Integer id;  
  
 *private* String name;  
  
 @Lob  
 *private* Clob warranty;  
  
 *//Getters and setters are omitted for brevity*}

Để persist một Entity, ta phải tạo một Clob sử dụng ClobProxy Hibernate utility:

String warranty = "My product warranty";  
  
*final* Product product = *new* Product();  
 product.setId( 1 );  
 product.setName( "Mobile phone" );  
  
 product.setWarranty( ClobProxy.generateProxy( warranty ) );  
  
 entityManager.persist( product );

Để lấy ra nội dung của Clob, ta cần phải chuyển đổi java.io.Reader gốc:

Product product = entityManager.find( Product.*class*, productId );  
  
 *try* (Reader reader = product.getWarranty().getCharacterStream()) {  
 assertEquals( "My product warranty", toString( reader ) );  
 }

Ngoài ra, @Lob còn có thể dùng để map với String và Char.

Mapping Blob cũng hoạt động tương tự, sử dụng @Lob, và map được với byte.

Tương tự, để persist một Entity, bạn phải tạo một Blob sử dụng BlobProxy Hibernate utility:

#### 2.3.9. Mapping Nationalized Character Data

JDBC 4 đã thêm khả năng xử lý rõ ràng dữ liệu ký tự được quốc hữu hóa. Để kết thúc này, nó đã thêm các kiểu dữ liệu ký tự được quốc hữu hóa cụ thể:

* NCHAR
* NVARCHAR
* LONGNVARCHAR
* NCLOB

Cách mapping là sử dụng Annotation @Nationalize với Nvarchar, char[] và Nclob.

Hibernate cũng có thể ánh xạ NCLOB tới một java.sql.NClob

#### 2.3.10. Mapping UUID Values

Hibernate cũng đồng thời cho phép bạn ánh xạ các giá trị UUID theo nhiều cách:

Note: ánh xạ UUID mặc định là nhị phân.

#### 2.3.11. UUID dạng nhị phân binary

Như đã đề cập, mapping thuộc tính UUID mặc định là nhị phân. . Ánh xạ UUID tới một  byte[] sử dụng  java.util.UUID#getMostSignificantBits và java.util.UUID#getLeastSignificantBits. Nó được lưu trữ như BINARY data.

Được chọn làm mặc định đơn giản vì nó thường hiệu quả hơn theo góc độ lưu trữ.

#### 2.3.12. UUID dạng (var)char

Maps UUID tới String sử dụng java.util.UUID#toString và java.util.UUID#fromString , lưu trữ như CHAR hay VARCHAR data.

#### 2.3.14. Dùng UUID như là identifier

Hibernate hỗ trợ sử dụng các giá trị UUID như những identifiers.

#### Mapping các giá trị Date/Time

Hibernate cho phép những lớp Java Date/Time ánh xạ thành các thuộc tính của persistent domain model entity. SQL cơ bản định nghĩa 3 loại sau:

DATE

Đại diện cho lịch lưu trữ năm, tháng, ngày. Tương đương với java.sql.Date trong JDBC

TIME

Đại diện cho thời gian một ngày và lưu giữ giờ phút giây, tương đương với java.sql.Time trong JDBC.

TIMESTAMP

Lưu giữ cả DATE và TIME cộng với nano giây. Tương đương với java.sql.Timestamp trong JDBC.

Các lớp java.sql xác định một liên kết trực tiếp tới kiểu dữ liệu SQL Date/Time, các thuộc tính  java.util hay java.time  cần được đánh dấu rõ ràng các mối tương quan kiểu SQL với @Temporal annotation. Bằng cách này, java.util.Date  hay một java.util.Calendar có thể ánh xạ thành một loại SQL DATE, TIME or TIMESTAMP type.

java.util.Date*mapped as*DATE

@Entity(name = "DateEvent")  
*public static class* DateEvent {  
  
 @Id  
 @GeneratedValue  
 *private* Long id;  
  
 @Column(name = "`timestamp`")  
 @Temporal(TemporalType.DATE)  
 *private* Date timestamp;  
  
 *//Getters and setters are omitted for brevity*}

*Persisting một*java.util.Date*mapping*

DateEvent dateEvent = *new* DateEvent( *new* Date() );  
 entityManager.persist( dateEvent );

SQL sản sinh ra là :

*INSERT INTO* DateEvent ( *timestamp*, id )  
*VALUES* ( '2015-12-29', 1 )

@Temporal có thể dùng Time và Date, Timestamp.

##### **Sử dụng một time zone chỉ định:**

Theo mặc định, Hibernate sẽ dùng timezone mặc định của JVM. Tuy nhiên ta có thể sửa nếu cần thiết.

#### 2.3.16. JPA 2.1 Bộ chuyển đổi thuộc tính - AttributeConverters

Mặc dù Hibernate có thể hỗ trợ [custom types](https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html" \l "basic-custom-type) nhưng nó cũng đồng thời hỗ trợ AttributeConverter.

Cùng với một AttributeConverter tùy chỉnh, chúng ta có thể ánh xạ kiểu JDBC tới một kiểu thực thể.

Ví dụ sau đây,  java.time.Period sẽ ánh xạ VARCHAR  tới cột trong database.

@Converter  
*public class* PeriodStringConverter  
 *implements* AttributeConverter<Period, String> {  
  
 @Override  
 *public* String convertToDatabaseColumn(Period attribute) {  
 *return* attribute.toString();  
 }  
  
 @Override  
 *public* Period convertToEntityAttribute(String dbData) {  
 *return* Period.parse( dbData );  
 }  
}

Để sử dụng Converter này, annotation @Converter sẽ được sử dụng.

@Entity(name = "Event")  
*public static class* Event {  
  
 @Id  
 @GeneratedValue  
 *private* Long id;  
  
 @Convert(converter = PeriodStringConverter.*class*)  
 @Column(columnDefinition = "")  
 *private* Period span;  
  
 *//Getters and setters are omitted for brevity*}

SQL sản sinh:

*INSERT INTO Event* ( span, id )  
*VALUES* ( 'P1Y2M3D', 1 )

##### **JPA 2.1 AttributeConverter Mutability Plan**

Một kiểu cơ bản được chuyển đổi bởi JPA AttributeConverter là bất biến nếu kiểu Java gốc cũng bất biến. Tương tự, nếu kiểu Java gốc là có thể thay đổi thì kiểu chuyển đổi bởi JPA cũng có thể thay đổi.

###### **Kiểu bất biến - Immutable types**

Nếu thuộc tính của thực thể là một String, một kiểu wrapper nguyên thủy như (e.g. Integer, Long), loại Enum, hay bất kỳ những kiểu Object bất biến nào khác, và bạn chỉ có thể thay đổi thuộc tính bằng cách giá trị khác.

Ở ví dụ sau, java.time.Period sẽ được map tới cột VARCHAR trong database.

@Entity(name = "Event")  
*public static class* Event {  
 @Id  
 @GeneratedValue  
 *private* Long id;  
 @Convert(converter = PeriodStringConverter.*class*)  
 @Column(columnDefinition = "")  
 *private* Period span;  
 *//Getters and setters are omitted for brevity*}

Cách duy nhất để thay đổi thuộc tính Span chính là đăng ký lại nó.

Event event = entityManager.createQuery( "from Event", Event.*class* ).getSingleResult();  
 event.setSpan(Period  
 .ofYears( 3 )  
 .plusMonths( 2 )  
 .plusDays( 1 )  
 );

###### **Kiểu khả biến - Mutable types**

Mặt khác, xét theo ví dụ sau khi Money là khả biến:

*public static class* Money {  
  
 *private long* cents;  
  
 *//Getters and setters are omitted for brevity*}  
  
@Entity(name = "Account")  
*public static class* Account {  
  
 @Id  
 *private* Long id;  
  
 *private* String owner;  
  
 @Convert(converter = MoneyConverter.*class*)  
 *private* Money balance;  
  
 *//Getters and setters are omitted for brevity*}  
  
*public static class* MoneyConverter  
 *implements* AttributeConverter<Money, Long> {  
  
 @Override  
 *public* Long convertToDatabaseColumn(Money attribute) {  
 *return* attribute == *null* ? *null* : attribute.getCents();  
 }  
  
 @Override  
 *public* Money convertToEntityAttribute(Long dbData) {  
 *return* dbData == *null* ? *null* : *new* Money( dbData );  
 }  
}

#### 2.3.17. SQL quoted identifiers

Bạn có thể buộc Hibernate đánh dấu một identifier trong SQL sinh ra bằng cách đặt tên bảng hay cột trong dấu nháy đơn. Theo truyền thống, Hibernate sử dụng dấu nháy đơn, còn JPA thì sử dụng dấu nháy kép.

Ngoại trừ những thứ trong dấu nháy, Hibernate sẽ thay đổi dựa trên SQL Dialect mà sản sinh ra SQL tương ứng.

*Hibernate legacy quoting*

@Entity(name = "Product")  
*public static class* Product {  
  
 @Id  
 *private* Long id;  
  
 @Column(name = "`name`")  
 *private* String name;  
  
 @Column(name = "`number`")  
 *private* String number;  
  
 *//Getters and setters are omitted for brevity*}

*JPA quoting*

@Entity(name = "Product")  
*public static class* Product {  
  
 @Id  
 *private* Long id;  
  
 @Column(name = "\"name\"")  
 *private* String name;  
  
 @Column(name = "\"number\"")  
 *private* String number;  
  
 *//Getters and setters are omitted for brevity*}

*Persisting một tên cột:*

Product product = *new* Product();  
 product.setId( 1L );  
 product.setName( "Mobile phone" );  
 product.setNumber( "123-456-7890" );  
 entityManager.persist( product );  
 INSERT INTO Product ("name", "number", id)  
 VALUES ('Mobile phone', '123-456-7890', 1)

Mã SQL sinh ra:

*INSERT INTO* Product ("name", "number", id)  
*VALUES* ('Mobile phone', '123-456-7890', 1)

##### **Global quoting**

Hibernate cũng có thể đánh dấu toàn bộ các identifiers (e.g. table, columns) sử dụng configuration property sau đây:

<property  
 *name*="hibernate.globally\_quoted\_identifiers"  
 *value*="true"  
/>

Như vậy, Hibernate sẽ mặc định đánh dấu indentifiers bằng tên thuộc tính trong dấu nháy, mà mã SQL sinh ra vẫn giống phần trên.

@Entity(name = "Product")  
*public static class* Product {  
  
 @Id  
 *private* Long id;  
  
 *private* String name;  
  
 *private* String number;  
  
 *//Getters and setters are omitted for brevity*}

#### 2.3.18. Thuộc tính sản sinh - Generated properties

Thuộc tính sản sinh (Generated properties) là các thuộc tính mà giá trị của nó được tạo ra bởi database. Thông thường, ứng dụng Hibernate cần refresh các đối tượng chứa bất kỳ thuộc tính nào mà cơ sở dữ liệu đang tạo giá trị. Tuy nhiên, việc đánh dấu các thuộc tính đó là thuộc tính sản sinh (generated) lại được chương trình ủy thác trách nhiệm cho Hibernate. Khi Hibernate phát hành SQL INSERT hay UPDATE cho một thực thể đã xác định sẵn thuộc tính được tạo. Nó ngay lập tức đưa ra một lựa chọn để truy xuất giá trị được tạo ra.

Các thuộc tính được đánh dấu là generated phải *không thể chèn* và *không thể cập nhật*. Chỉ có kiểu @Version and @Basic là được đánh giá là generated.

NEVER (mặc định)

Giá trị thuộc tính được cho không sản sinh trong database.

INSERT

Giá trị thuộc tính được tạo ra khi chèn nhưng không được tạo lại một lần nữa trong các bản cập nhật tiếp theo. Các thuộc tính như creationTimestamp rơi vào danh mục này.

ALWAYS

Giá trị thuộc tính sẽ được sản sinh cả khi chèn lẫn cập nhật.

Để đánh dấu một thuộc tính sản sinh (generated) Hibernate sử dụng @Generated annotation.

##### **@Generated annotation**

[@Generated](https://docs.jboss.org/hibernate/orm/5.3/javadocs/org/hibernate/annotations/Generated.html) annotation được sử dụng giúp Hibernate có thể tìm nạp thuộc tính đã được chú thích ngay cả sau khi thực thể đã persisted hay updated. Vì lý do này, @Generated annotation chấp nhận một giá trị enum [GenerationTime](https://docs.jboss.org/hibernate/orm/5.3/javadocs/org/hibernate/annotations/GenerationTime.html).

@Generated*ví dụ:*

@Entity(name = "Person")  
*public static class* Person {  
  
 @Id  
 *private* Long id;  
  
 *private* String firstName;  
  
 *private* String lastName;  
  
 *private* String middleName1;  
  
 *private* String middleName2;  
  
 *private* String middleName3;  
  
 *private* String middleName4;  
  
 *private* String middleName5;  
  
 @Generated( value = GenerationTime.ALWAYS )  
 @Column(columnDefinition =  
 "AS CONCAT(" +  
 " COALESCE(firstName, ''), " +  
 " COALESCE(' ' + middleName1, ''), " +  
 " COALESCE(' ' + middleName2, ''), " +  
 " COALESCE(' ' + middleName3, ''), " +  
 " COALESCE(' ' + middleName4, ''), " +  
 " COALESCE(' ' + middleName5, ''), " +  
 " COALESCE(' ' + lastName, '') " +  
 ")")  
 *private* String fullName;  
  
}

Khi thực thể Person đã được persist, Hibernate sẽ tìm nạp cột  fullName từ database, nối liền first, middle, và last name.

@Generated*persist example*

Person person = *new* Person();  
 person.setId( 1L );  
 person.setFirstName( "John" );  
 person.setMiddleName1( "Flávio" );  
 person.setMiddleName2( "André" );  
 person.setMiddleName3( "Frederico" );  
 person.setMiddleName4( "Rúben" );  
 person.setMiddleName5( "Artur" );  
 person.setLastName( "Doe" );  
  
 entityManager.persist( person );  
 entityManager.flush();  
  
 assertEquals("John Flávio André Frederico Rúben Artur Doe", person.getFullName());

Mã SQL:

*INSERT INTO* Person  
(  
 firstName,  
 lastName,  
 middleName1,  
 middleName2,  
 middleName3,  
 middleName4,  
 middleName5,  
 id  
)  
*values*(?, ?, ?, ?, ?, ?, ?, ?)  
  
*-- binding parameter [1] as [VARCHAR] - [John]  
-- binding parameter [2] as [VARCHAR] - [Doe]  
-- binding parameter [3] as [VARCHAR] - [Flávio]  
-- binding parameter [4] as [VARCHAR] - [André]  
-- binding parameter [5] as [VARCHAR] - [Frederico]  
-- binding parameter [6] as [VARCHAR] - [Rúben]  
-- binding parameter [7] as [VARCHAR] - [Artur]  
-- binding parameter [8] as [BIGINT] - [1]  
  
SELECT* p.fullName *as* fullName3\_0\_  
*FROM* Person p  
*WHERE* p.id=?  
  
*-- binding parameter [1] as [BIGINT] - [1]  
-- extracted value ([fullName3\_0\_] : [VARCHAR]) - [John Flávio André Frederico Rúben Artur Doe]*

Điều này cũng diễn ra tương tự khi update. Hibernate sẽ tìm nạp lại cột fullName từ database sau khi thực thể được thay đổi.

@Generated*ví dụ về update*

Person person = entityManager.find( Person.*class*, 1L );  
 person.setLastName( "Doe Jr" );  
  
 entityManager.flush();  
 assertEquals("John Flávio André Frederico Rúben Artur Doe Jr", person.getFullName());

Mã SQL sinh ra:

*UPDATE* Person  
*SET* firstName=?,  
 lastName=?,  
 middleName1=?,  
 middleName2=?,  
 middleName3=?,  
 middleName4=?,  
 middleName5=?  
*WHERE* id=?  
  
*-- binding parameter [1] as [VARCHAR] - [John]  
-- binding parameter [2] as [VARCHAR] - [Doe Jr]  
-- binding parameter [3] as [VARCHAR] - [Flávio]  
-- binding parameter [4] as [VARCHAR] - [André]  
-- binding parameter [5] as [VARCHAR] - [Frederico]  
-- binding parameter [6] as [VARCHAR] - [Rúben]  
-- binding parameter [7] as [VARCHAR] - [Artur]  
-- binding parameter [8] as [BIGINT] - [1]  
  
SELECT* p.fullName *as* fullName3\_0\_  
*FROM* Person p  
*WHERE* p.id=?  
  
*-- binding parameter [1] as [BIGINT] - [1]  
-- extracted value ([fullName3\_0\_] : [VARCHAR]) - [John Flávio André Frederico Rúben Artur Doe Jr]*

##### **@GeneratorType annotation**

[@GeneratorType](https://docs.jboss.org/hibernate/orm/5.3/javadocs/org/hibernate/annotations/GeneratorType.html) annotation được sử dụng để cung cấp một custom generator cho phép thiết lập một giá trị cho thuộc tính (property) được chú thích.

Vì lý do này, @GeneratorType annotation chấp nhận một  [GenerationTime](https://docs.jboss.org/hibernate/orm/5.3/javadocs/org/hibernate/annotations/GenerationTime.html) có giá trị enum và một kiểu class custom [ValueGenerator](https://docs.jboss.org/hibernate/orm/5.3/javadocs/org/hibernate/annotations/ValueGenerator.html) .

@GeneratorType*mapping example*

*public static class* CurrentUser {  
  
 *public static final* CurrentUser INSTANCE = *new* CurrentUser();  
  
 *private static final* ThreadLocal<String> storage = *new* ThreadLocal<>();  
  
 *public void* logIn(String user) {  
 storage.set( user );  
 }  
  
 *public void* logOut() {  
 storage.remove();  
 }  
  
 *public* String get() {  
 *return* storage.get();  
 }  
}  
  
*public static class* LoggedUserGenerator *implements* ValueGenerator<String> {  
  
 @Override  
 *public* String generateValue(  
 Session session, Object owner) {  
 *return* CurrentUser.INSTANCE.get();  
 }  
}  
  
@Entity(name = "Person")  
*public static class* Person {  
  
 @Id  
 *private* Long id;  
  
 *private* String firstName;  
  
 *private* String lastName;  
  
 @GeneratorType( type = LoggedUserGenerator.*class*, when = GenerationTime.INSERT)  
 *private* String createdBy;  
  
 @GeneratorType( type = LoggedUserGenerator.*class*, when = GenerationTime.ALWAYS)  
 *private* String updatedBy;  
  
}

Khi thực thể Person được persisted, Hibernate sẽ điền vào cột createdBy với giá trị là người dùng đang đăng nhập.

@Generated*persist example*

CurrentUser.INSTANCE.logIn( "Alice" );  
  
 doInJPA( *this*::entityManagerFactory, entityManager -> {  
  
 Person person = *new* Person();  
 person.setId( 1L );  
 person.setFirstName( "John" );  
 person.setLastName( "Doe" );  
  
 entityManager.persist( person );  
 } );  
  
 CurrentUser.INSTANCE.logOut();

Mã SQL sinh ra:

*INSERT INTO* Person  
(  
 createdBy,  
 firstName,  
 lastName,  
 updatedBy,  
 id  
)  
*VALUES*(?, ?, ?, ?, ?)  
  
*-- binding parameter [1] as [VARCHAR] - [Alice]  
-- binding parameter [2] as [VARCHAR] - [John]  
-- binding parameter [3] as [VARCHAR] - [Doe]  
-- binding parameter [4] as [VARCHAR] - [Alice]  
-- binding parameter [5] as [BIGINT] - [1]*

Điều tương tự cũng diễn ra khi thực thể Person được update. Hibernate sẽ điền vào cột the updatedBy bằng giá trị người dùng đăng nhập.

@Generated*update example*

CurrentUser.INSTANCE.logIn( "Bob" );  
  
 doInJPA( *this*::entityManagerFactory, entityManager -> {  
 Person person = entityManager.find( Person.*class*, 1L );  
 person.setFirstName( "Mr. John" );  
 } );  
  
 CurrentUser.INSTANCE.logOut();

Mã SQL sinh ra:

*UPDATE* Person  
*SET* createdBy = ?,  
 firstName = ?,  
 lastName = ?,  
 updatedBy = ?  
*WHERE* id = ?  
  
*-- binding parameter [1] as [VARCHAR] - [Alice]  
-- binding parameter [2] as [VARCHAR] - [Mr. John]  
-- binding parameter [3] as [VARCHAR] - [Doe]  
-- binding parameter [4] as [VARCHAR] - [Bob]  
-- binding parameter [5] as [BIGINT] - [1]*

@CreationTimestamp*mapping example*

@Entity(name = "Event")  
*public static class* Event {  
  
 @Id  
 @GeneratedValue  
 *private* Long id;  
  
 @Column(name = "`timestamp`")  
 @CreationTimestamp  
 *private* Date timestamp;  
  
 *//Constructors, getters, and setters are omitted for brevity*}

Khi thực thể Event được tạo, Hibernate sẽ điền vào cột timestamp với giá trị thời gian hiện tại của timestamp.

@CreationTimestamp*persist example*

Event dateEvent = *new* Event( );  
 entityManager.persist( dateEvent );

Mã SQL:

*INSERT INTO Event* ("timestamp", id)  
*VALUES* (?, ?)  
  
*-- binding parameter [1] as [TIMESTAMP] - [Tue Nov 15 16:24:20 EET 2016]  
-- binding parameter [2] as [BIGINT] - [1]*

##### **@UpdateTimestamp annotation**

 @UpdateTimestamp annotation chỉ thị Hibernate thiết lập một thuộc tính của thực thể được chú thích (annotated entity attribute) với giá trị hiện tại của JVM timestamp khi thực thể này được persist.

Những kiểu thuộc tính được hỗ trợ là:

* java.util.Date
* java.util.Calendar
* java.sql.Date
* java.sql.Time
* java.sql.Timestamp

@UpdateTimestamp*mapping example*

@Entity(name = "Bid")  
*public static class* Bid {  
  
 @Id  
 @GeneratedValue  
 *private* Long id;  
  
 @Column(name = "updated\_on")  
 @UpdateTimestamp  
 *private* Date updatedOn;  
  
 @Column(name = "updated\_by")  
 *private* String updatedBy;  
  
 *private* Long cents;  
  
 *//Getters and setters are omitted for brevity*}

Khi thực thể Bid được persisted, Hibernate sẽ chèn vào cột gốc  updated\_on với giá trị timestamp của JVM.

@UpdateTimestamp*persist example*

Bid bid = *new* Bid();  
 bid.setUpdatedBy( "John Doe" );  
 bid.setCents( 150 \* 100L );  
 entityManager.persist( bid );

Mã SQL sinh ra:

*INSERT INTO* Bid (cents, updated\_by, updated\_on, id)  
*VALUES* (?, ?, ?, ?)  
  
*-- binding parameter [1] as [BIGINT] - [15000]  
-- binding parameter [2] as [VARCHAR] - [John Doe]  
-- binding parameter [3] as [TIMESTAMP] - [Tue Apr 18 17:21:46 EEST 2017]  
-- binding parameter [4] as [BIGINT] - [1]*

*@UpdateTimestamp update example*

When updating the Bid entity, Hibernate is going to modify the updated\_on column with the current JVM timestamp value:

Bid bid = entityManager.find( Bid.*class*, 1L );  
  
 bid.setUpdatedBy( "John Doe Jr." );  
 bid.setCents( 160 \* 100L );  
 entityManager.persist( bid );

SQL:

*UPDATE* Bid *SET* cents = ?,  
 updated\_by = ?,  
 updated\_on = ?  
*where* id = ?  
  
*-- binding parameter [1] as [BIGINT] - [16000]  
-- binding parameter [2] as [VARCHAR] - [John Doe Jr.]  
-- binding parameter [3] as [TIMESTAMP] - [Tue Apr 18 17:49:24 EEST 2017]  
-- binding parameter [4] as [BIGINT] - [1]*

##### **@ValueGenerationType meta-annotation**

Hibernate 4.3 giới thiệu @ValueGenerationType meta-annotation, đây là một cách tiếp cận mới để khai báo các thuộc tính được tạo hoặc tùy chỉnh các trình được tạo.

@Generated đã được trang bị thêm để sử dụng @ValueGenerationType meta-annotation. Nhưng @ValueGenerationType hiển thị nhiều tính năng hơn những gì @Generated  hỗ trợ. Để tận dụng những tính năng đó, bạn đơn giản cần kết nối với một generator annotation mới.

Như bạn sẽ thấy trong ví dụ sau đây, @ValueGenerationType meta-annotation được sử dụng khi khai báo custom annotation được sử dụng để đánh dấu các thuộc tính thực thể (entity properties) mà cần một chiến lược khởi tạo cụ thể. Logic thế hệ thực (actual generation logic) cần phải được thêm vào lớp thực hiện  AnnotationValueGeneration interface.

###### **Database-generated values**

Để ví dụ, hãy nói rằng chúng ta muốn timestamps được tạo ra bằng cách gọi một hàm ANSI SQL cơ bản current\_timestamp (thay vì gọi triggers hay giá trị DEFAULT :

ValueGenerationType*mapping for database generation*

@Entity(name = "Event")  
*public static class* Event {  
  
 @Id  
 @GeneratedValue  
 *private* Long id;  
  
 @Column(name = "`timestamp`")  
 @FunctionCreationTimestamp  
 *private* Date timestamp;  
  
 *//Constructors, getters, and setters are omitted for brevity*}  
  
@ValueGenerationType(generatedBy = FunctionCreationValueGeneration.*class*)  
@Retention(RetentionPolicy.RUNTIME)  
*public* @*interface* FunctionCreationTimestamp {}  
  
*public static class* FunctionCreationValueGeneration  
 *implements AnnotationValueGeneration*<FunctionCreationTimestamp> {  
  
 @Override  
 *public void* initialize(FunctionCreationTimestamp annotation, Class<?> propertyType) {  
 }  
  
 */\*\*  
 \* Generate value on INSERT  
 \** ***@return*** *when to generate the value  
 \*/  
 public* GenerationTiming getGenerationTiming() {  
 *return* GenerationTiming.INSERT;  
 }  
  
 */\*\*  
 \* Returns null because the value is generated by the database.  
 \** ***@return*** *null  
 \*/  
 public ValueGenerator*<?> getValueGenerator() {  
 *return null*;  
 }  
  
 */\*\*  
 \* Returns true because the value is generated by the database.  
 \** ***@return*** *true  
 \*/  
 public boolean* referenceColumnInSql() {  
 *return true*;  
 }  
  
 */\*\*  
 \* Returns the database-generated value  
 \** ***@return*** *database-generated value  
 \*/  
 public* String getDatabaseGeneratedReferencedColumnValue() {  
 *return* "current\_timestamp";  
 }  
}

Khi persist thực thể Event, ta được mã SQL

*INSERT INTO Event* ("timestamp", id)  
*VALUES* (*current\_timestamp*, 1)

Như đã thấy, giá trị  current\_timestamp đã được gán cho giá trị của cột timestamp .

In-memory-generated values

Nếu giá trị timestamp cần phải được tạo ra từ bộ nhớ trong. Ta sử dụng phương pháp sau:

@Entity(name = "Event")  
*public static class* Event {  
  
 @Id  
 @GeneratedValue  
 *private* Long id;  
  
 @Column(name = "`timestamp`")  
 @FunctionCreationTimestamp  
 *private* Date timestamp;  
  
 *//Constructors, getters, and setters are omitted for brevity*}  
  
@ValueGenerationType(generatedBy = FunctionCreationValueGeneration.*class*)  
@Retention(RetentionPolicy.RUNTIME)  
*public* @*interface* FunctionCreationTimestamp {}  
  
*public static class* FunctionCreationValueGeneration  
 *implements* AnnotationValueGeneration<FunctionCreationTimestamp> {  
  
 @Override  
 *public void* initialize(FunctionCreationTimestamp annotation, Class<?> propertyType) {  
 }  
  
 */\*\*  
 \* Generate value on INSERT  
 \** ***@return*** *when to generate the value  
 \*/  
 public* GenerationTiming getGenerationTiming() {  
 *return* GenerationTiming.INSERT;  
 }  
  
 */\*\*  
 \* Returns the in-memory generated value  
 \** ***@return*** *{****@code*** *true}  
 \*/  
 public* ValueGenerator<?> getValueGenerator() {  
 *return* (session, owner) -> *new* Date( );  
 }  
  
 */\*\*  
 \* Returns false because the value is generated by the database.  
 \** ***@return*** *false  
 \*/  
 public boolean* referenceColumnInSql() {  
 *return false*;  
 }  
  
 */\*\*  
 \* Returns null because the value is generated in-memory.  
 \** ***@return*** *null  
 \*/  
 public* String getDatabaseGeneratedReferencedColumnValue() {  
 *return null*;  
 }  
}

#### 2.3.19. Column transformers: read and write expressions

Hibernate cho phép bạn tùy chỉnh SQL mà nó sử dụng để đọc và ghi các giá trị của các cột được ánh xạ tới các @Basic type

. Ví dụ: nếu cơ sở dữ liệu của bạn cung cấp một tập hợp các hàm mã hóa dữ liệu, bạn có thể gọi chúng cho các cột riêng lẻ như trong ví dụ sau.

@ColumnTransformer*example*

@Entity(name = "Employee")  
*public static class* Employee {  
  
 @Id  
 *private* Long id;  
  
 @NaturalId  
 *private* String username;  
  
 @Column(name = "pswd")  
 @ColumnTransformer(  
 read = "decrypt( 'AES', '00', pswd )",  
 write = "encrypt('AES', '00', ?)"  
 )  
 *private* String password;  
  
 *private int* accessLevel;  
  
 @ManyToOne(fetch = FetchType.LAZY)  
 *private* Department department;  
  
 @ManyToMany(mappedBy = "employees")  
 *private* List<Project> projects = *new* ArrayList<>();  
  
 *//Getters and setters omitted for brevity*}

@ColumnTransformerforColumn*attribute usage*

@Entity(name = "Savings")  
*public static class* Savings {  
  
 @Id  
 *private* Long id;  
  
 @Type(type = "org.hibernate.userguide.mapping.basic.MonetaryAmountUserType")  
 @Columns(columns = {  
 @Column(name = "money"),  
 @Column(name = "currency")  
 })  
 @ColumnTransformer(  
 forColumn = "money",  
 read = "money / 100",  
 write = "? \* 100"  
 )  
 *private* MonetaryAmount wallet;  
  
 *//Getters and setters omitted for brevity*}

Hibernate tự động áp dụng các biểu thức tùy chỉnh bất cứ khi nào thuộc tính được tham chiếu trong truy vấn. Chức năng này tương tự như @Formula với hai điểm khác biệt:

* Thuộc tính được hỗ trợ bởi một hoặc nhiều cột được xuất như một phần của việc tạo lược đồ tự động.
* Tài sản là đọc-ghi, không chỉ đọc.