# Оглавление

[Оглавление 1](#_Toc56755422)

[1. HIBERNATE 3](#_Toc56755423)

[2. MAPPING TYPES 4](#_Toc56755424)

[2.1. VALUE TYPES (BASIC) 4](#_Toc56755425)

[2.1.1. MAPPING ENUMS 6](#_Toc56755426)

[2.1.2. ATTRIBUTE CONVERTOR 7](#_Toc56755427)

[2.1.3. MAPPING LARGE OBJECTS (LOB) 7](#_Toc56755428)

[2.1.4. MAPPING NATIONALIZED CHARACTER DATA 7](#_Toc56755429)

[2.1.5. MAPPING UUID VALUES 8](#_Toc56755430)

[2.1.6. MAPPING DATA/TIME VALUES 8](#_Toc56755431)

[2.2. VALUE TYPES (EMBEDDABLE) 8](#_Toc56755432)

[3. ENTITY TYPES 9](#_Toc56755433)

[3.1. Implementing equals() and hashCode() 10](#_Toc56755434)

[3.2. Mapping the entity to a SQL query 10](#_Toc56755435)

[3.3. Define a custom entity proxy 10](#_Toc56755436)

[3.4. Define a custom entity persister 10](#_Toc56755437)

[3.5. Access strategies 10](#_Toc56755438)

[3.6. Identifiers 11](#_Toc56755439)

[3.6.1. Simple identifiers 11](#_Toc56755440)

[3.6.2. Compositeidentifiers 11](#_Toc56755441)

[3.6.3. Derivedidentifiers 12](#_Toc56755442)

[3.6.4. @RowId 12](#_Toc56755443)

[3.7. Generated identifier values 12](#_Toc56755444)

[3.7.1. Identity generation type 12](#_Toc56755445)

[3.7.2. Sequence generation type 12](#_Toc56755446)

[3.8. Optimizers 12](#_Toc56755447)

[3.9. @GenericGenerator 13](#_Toc56755448)

[4. GENERATEDPROPERTIES 13](#_Toc56755449)

[5. COLUMN TRANSFORMERS: READ AND WRITE EXPRESSIONS 14](#_Toc56755450)

[6. NAMINGSTRATEGIES 14](#_Toc56755451)

[7. SQL QUOTEDIDENTIFIERS 16](#_Toc56755452)

[8. ASSOCIATIONS 16](#_Toc56755453)

[8.1. ManyToOne 16](#_Toc56755454)

[8.2. OneToMany 16](#_Toc56755455)

[8.3. OneToOne 17](#_Toc56755456)

[8.4. ManyToMany 17](#_Toc56755457)

[8.5. ManyToOne + Any 18](#_Toc56755458)

[8.6. ManyToAny 18](#_Toc56755459)

[8.7. ManyToOne + JoinFormula 18](#_Toc56755460)

[8.8. ManyToOne + JoinColumnOrFormula 18](#_Toc56755461)

[9. COLLECTIONS 18](#_Toc56755462)

[9.1. Collections of value types 19](#_Toc56755463)

[9.2. Collections of entities 19](#_Toc56755464)

[9.2.1. Bags 19](#_Toc56755465)

[9.2.2. Ordered lists 20](#_Toc56755466)

[9.2.3. Sets 20](#_Toc56755467)

[9.2.4. Maps 21](#_Toc56755468)

[9.2.5. Arrays 21](#_Toc56755469)

[9.2.6. Collection as basic value type 21](#_Toc56755470)

[9.2.7. Custom collection type 21](#_Toc56755471)

[10. NATURAL IDS 22](#_Toc56755472)

[11. DYNAMIC MODEL 22](#_Toc56755473)

[12. INHERITANCE 22](#_Toc56755474)

[Polymorphic Queries 23](#_Toc56755475)

[13. IMMUTABILITY 23](#_Toc56755476)

[14. BOOTSTRAP 23](#_Toc56755477)

[15. ССЫЛКИ 23](#_Toc56755478)

# HIBERNATE

**Hibernate** – этоJPA-провайдер, которыйреализует (JavaPersistenceAPI)**ирасширяет!**(Hibernate Native API - HNA) интерфейс JPA.



Рисунок 1.1Структура приложения, использующего Hibernate.

**Примечание**: для того, чтобы иметь возможность впоследствии безболезненно сменить JPA-провайдера, необходимо использовать только JPA (без HNA).



Рисунок 1.2 Структура приложения, использующего Hibernate.

На рисунке выше желтым цветом отмечены классы и интерфейсы, относящиеся к JPA,серым – классы и интерфейсы, относящиеся к HNA.

**SessionFactory** – потокобезопасное (неизменяемое) представление мапинга доменной модели приложения на базу данных. Является фабрикой для создания объектов org.hibernate.Session. EntityManagerFactory – это JPA эквивалент SessionFactory.У под капотомSessionFactory (не точно).

SessionFactory требует много ресурсов при создании, поэтому в приложении должен быть инстанциирован только один экземпляр этого класса.

SessionFactory поддерживает сервисы, которые Hibernate использует сквозь все Sessions:

* Second level cashes;
* Connection pools;
* Transaction system integrations.

**Session** – однопоточный короткоживущий объект (“Unitofwork”). EntityManager – это JPA эквивалент Session. Session по сути является обверткой над java.sql.Connection и действует как фабрика для создания объектов типа org.hibernate.Transaction.

С каждым объектом сессии ассоциируется “Repeatable read” persistence context (First level cash). [http://learningviacode.blogspot.com/2012/02/first-level-cache-and-repeatable-reads.html - краткое введение](http://learningviacode.blogspot.com/2012/02/first-level-cache-and-repeatable-reads.html%20-%20краткое%20введение). (Т.е. в рамках одной сессии для конкретного Entity результат запроса будет возвращать один и тот же объект (?)).

**Transaction** – однопоточный короткоживущий объект, используемый приложением для разграничения индивидуальных физических границ транзакций (EntityTransaction – JPA эквивалент). Оба класса действуют как абстрактный API для изолирования приложения от underlyingtransactionsysteminuse (JDBCorJTA - https://laliluna.com/jpa-hibernate-guide/ch13s02.html).

# MAPPING TYPES

Hibernatetype не является ни Java типом, ни SQL типом. Hibernatetype предоставляет информацию о том, как отобразить Javatype на SQLtype, икак сохранить/извлечь Javatype в базу/из базы.

В широком смысле Hibernate разделяет типы на две группы:

1. Value types. (Relate to properties of persistence classes).
2. Entity types. (Relate to persistence classes).

Valuetypes в свою очередь делятся на:

1. Basic types.
2. Embeddable types.
3. Collection types.

# VALUE TYPES (BASIC)

Basic value types usually map a single database column, to a single, non-aggregated Java type. Internally Hibernate uses a registry of basic types when it needs to resolve a specific org.hibernate.type.Type.

| **Hibernate type (org.hibernate.type package)** | **JDBC type**  **(general SQL)** | **Javatype** | **BasicTypeRegistrykey(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 | character, char, java.lang.Character |
| BooleanType | BOOLEAN | 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 |
| **…** | **…** | **…** | **…** |

Аннотация **@basic** определяет базовые типы, и так как она является аннотацией по умолчанию, то определять ее для базовых типов не нужно.**The@Basicannotationonafieldorapropertysignifiesthatit'sabasictypeandHibernateshouldusethestandardmappingforitspersistence.**

**ЗавыборподходящегоHibernate-типаотвечает**org.hibernate.type.BasicTypeRegistry**, которыйпозволяет регистрировать новые пользовательские типы, и переопределять существующие.**

Явно задать тип можно с помощью аннотации org.hibernate.annotations.Type:

@org.hibernate.annotations.Type( type = "nstring" )

privateStringname;

которая в качестве атрибута может использовать:

1. Fully qualified name of any org.hibernate.type.Type implementation.
2. Any key registered with BasicTypeRegistry.
3. The name of any known *type definitions.*

Список классов, которые по спецификации JPAотносятся к базовым:

* Java primitive types (boolean, int, etc);
* wrappers for the primitive types (java.lang.Boolean, java.lang.Integer, etc);
* java.lang.String;
* java.math.BigInteger;
* java.math.BigDecimal;
* java.util.Date;
* java.util.Calendar;
* java.sql.Date;
* java.sql.Time;
* java.sql.Timestamp;
* byte[] or Byte[];
* char[] or Character[];
* enums;
* any other type that implements Serializable (JPA’s "support" for Serializable types is to directly serialize their state to the database).

Атрибутыаннотации@Basic:

**optional** - boolean (defaults to true)

Defines whether this attribute allows nulls. JPA defines this as "a hint", which essentially means that its effect is specifically required. As long as the type is not primitive, Hibernate takes this to mean that the underlying column should be NULLABLE.

**fetch** - FetchType (defaults to EAGER)

Defines whether this attribute should be fetched eagerly or lazily. JPA says that EAGER is a requirement to the provider (Hibernate) that the value should be fetched when the owner is fetched, while LAZY is merely a hint that the value is fetched when the attribute is accessed. Hibernate ignores this setting for basic types unless you are using bytecode enhancement.

Создать собственный hibernateтип можно двумя способами:

* Implementing a BasicType and registering it;
* Implementing a UserType, which doesn’t require type registration.

Использовать пользовательский базовый тип можно через аннотацию org.hibernate.annotations.Type с указанием полного имени. Также можно зарегистрировать новый созданный тип, что не потребует указания полного имени. Дополнительно класс можно аннотировать:

@TypeDef(

name = "bitset",

defaultForType = BitSet.class,

typeClass = BitSetType.class

)

В этом случае регистрация так же не требуется.

Способы регистрации: …

Примечание: можно создавать типы, которые, например, будут сохранять одно поле в несколько колонок(<https://www.baeldung.com/hibernate-custom-types>).

# MAPPING ENUMS

Hibernate позволяет отображать Enum-тип в качестве базового. Для этого используется аннотация @Enumeratedс параметром EnumType.ORDINAL или EnumType.STRING. В первом случае поле будет сохранено как число, во втором как строка.

Можно также замапить Enumс помощью пользовательского типа (customtypes).

# ATTRIBUTE CONVERTOR

Attributeconventerпозволяет преобразовывать сохраняемое/извлекаемое значение. Например, еслиунасесть enumGender (MALE, FEMALE), товбазуможносохранять “F” и “M”соответственно.(JPA compatibility: JPA explicitly disallows the use of an AttributeConverter with an attribute marked as @Enumerated.)

The AttributeConverter entity property can be usedas a query parameter (<https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#basic-enums>).

AttributeConverterможет быть использован и при HBMмаппинге.

# MAPPING LARGE OBJECTS (LOB)

Для мапингаLOBможно использовать:

1. JDBC locator types.
2. Typesthat materializing the LOB data.

Materialized deals with the entire LOB contents in memory, whereas LOB locators (in theory) allow streaming parts of the LOB contents into memory as needed. A LOB locator is only portably valid during the duration of the transaction in which it was obtained.

The JDBC LOB locator types include (The JPA specification doesn’t define LOB locators):

* java.sql.Blob
* java.sql.Clob
* java.sql.NClob

Mapping materialized forms of these LOB values would use more familiar Java types such as

* String
* char[]
* byte[]
* etc.

Addition read - <https://www.youtube.com/watch?v=uZXfZZ59cjU>

# MAPPING NATIONALIZED CHARACTER DATA

JDBC4 поддерживает работу с национализированными символьными данными. Дляэтогоиспользуетсяаннотация@Nationalized.

If you application and database are entirely nationalized you may instead want to enable nationalized character data as the default. You can do this via the hibernate.use\_nationalized\_character\_data setting or by calling MetadataBuilder#enableGlobalNationalizedCharacterDataSupport during bootstrap.

# MAPPING UUID VALUES

…

# MAPPING DATA/TIME VALUES

Для мапинга даты можно использховать:

1. Классыдатыивремениизпакетаjava.sql.\*.
2. Классыдатыивремениизпакетаjava.util.\* (Date и Calendar).
3. Java 8 Data/Time.

SQLстандартопределяеттритипаDate/Time:

1. **DATE** - Represents a calendar date by storing years, months and days. The JDBC equivalent is java.sql.Date.
2. **TIME** - Represents the time of a day and it stores hours, minutes and seconds. The JDBC equivalent is java.sql.Time.
3. **TIMESTAMP** - It stores both a DATE and a TIME plus nanoseconds. The JDBC equivalent is java.sql.Timestamp.

Классы**java.util.Date**и**java.util.Calendar**неимеютоднозначногосоответствияsql-типамбазыданых, апоэтомудолжныспецифициоватьсядополнительнойаннотацией**@Temporal**суказаниемтипавпараметре:

* TemporalType.DATE;
* TemporalType.TIME;
* TemporalType.TIMESTAMP.

Классы из пакета java.time (Java8), которые моут быть использованы:

1. **DATE**- java.time.LocalDate.
2. **TIME**- java.time.LocalTime, java.time.OffsetTime.
3. **TIMESTAMP**- java.time.Instant, java.time.LocalDateTime, java.time.OffsetDateTime and java.time.ZonedDateTime.

Настройки временной зоны можно установить с помощью настроек, на уровне session или SessionFactory.

# VALUE TYPES (EMBEDDABLE)

Embeddabletypes или componentsподдерживаютконцепциюсоставныхзначений. В javaмодели классов может быть класс, который кроме обычных полей может состоять из других классов (embeddable). Эти embeddable классы мапятся в ту же таблицу, что и основной класс.

JPA defines two terms for working with an embeddable type: @Embeddable and @Embedded.

* @Embeddable is used to describe the mapping type itself (e.g. Publisher).
* @Embedded is for referencing a given embeddable type (e.g. book#publisher).

Использование нескольких embeddable классов одного и того же типа в классе допустимо, но в этом случае необходимо разрешить конфликты имен.

Это можно сделать:

* (Явно) использую аннотации [@AttributeOverride](http://docs.oracle.com/javaee/7/api/javax/persistence/AttributeOverride.html)and[@AssociationOverride](http://docs.oracle.com/javaee/7/api/javax/persistence/AssociationOverride.html)
* (Неявно) Использовать ImplicitNamingStrategy со значением *component-path.* (ImplicitNamingStrategy не поддерживается спецификацией JPA).

# ENTITY TYPES

Требования JPA-спецификации к entityclass:

1. Класс должен быть обозначен аннотацией @javax.persistnce.Entity или описан в xml-файле.
2. Класс должен иметь public или protectedno-argument конструктор. Дополнительные конструкторы разрешены.
3. Класс должен быть top-levelclass.
4. Интерфейс или enum не может быть entity классом.
5. Класс не должен быть final. Класс не должен иметь final methods или persistent instance variables.
6. If an entity instance is to be used remotely as a detached object, the entity class must implement the Serializable interface.
7. Both abstract and concrete classes can be entities. Entities may extend non-entity classes as well as entity classes, and non-entity classes may extend entity classes.
8. The persistent state of an entity is represented by instance variables, which may correspond to JavaBean-style properties. An instance variable must be directly accessed only from within the methods of the entity by the entity instance itself. The state of the entity is available to clients only through the entity’s accessor methods (getter/setter methods) or other business methods.

Однако, Hibernateне требует жестко следовать спецификации JPA:

1. Entityклассможетиметьno-argumentconstructor, которыйможетиметьpublic, protectedилиpackageвидимость. Допускаются дополнительные конструкторы.
2. Entityкласс не обязательно должен бытьtop-levelкласс.
3. Технически, Hibernate может persist final классыиликлассысfinal persistent state accessor (getter/setter) методами. However, it is generally not a good idea as doing so will stop Hibernate from being able to generate proxies for lazy-loading the entity.
4. Hibernate does not restrict the application developer from exposing instance variables and reference them from outside the entity class itself. The validity of such a paradigm, however, is debatable at best.

The main piece in mapping the entity is the javax.persistence.Entity annotation. Аннотация@Entityопределяеттолькоодинаттрибутname,которыйдает конкретной Entityимя, которое будет использовано в JPQLзапросах. Поумолчанию, имяentityпредставляетизсебяимясамого класса entity.

Anentitymodelsadatabasetable. The identifier uniquely identifies each row in that table. By default, the name of the table is assumed to be the same as the name of the entity. To explicitly give the name of the table or to specify other information about the table, we would use the javax.persistence.Table annotation.

# Implementing equals() and hashCode()

Hibernate guarantees equivalence of persistent identity (database row) and Java identity inside a particular session scope. Therefore, if we ask a Hibernate Session to load that specific Person multiple times we will actually get back the same *instance*:

Book book1 = entityManager.find(Book.class, 1L);

Book book2 = entityManager.find(Book.class, 1L);

book1 == book2;// true

Если book1 и book2 получены в разных сессиях, то будет возвращено два разных объекта.

book1 == book2;// false

# Mapping the entity to a SQL query

Hibernate позволяет мапить entityкласс на подзапрос с помощью аннотации @SubSelect. Таблица в базе данных в этом случае не нужна.

# Define a custom entity proxy

Hibernate позволяет использовать механизм прокси для final классов с помощью аннотации @Proxy(proxyClass = SomeProxyClass.class).

Так же можно создавать динамические entity proxy:

It is possible to map your entities as dynamic proxies using the [@Tuplizer](https://docs.jboss.org/hibernate/orm/5.3/javadocs/org/hibernate/annotations/Tuplizer.html) annotation.

# Define a custom entity persister

Hibernate позволяет создавать свои реализации классов, которые будут сохранять Entitiesи Collections в БД.

@Entity

@Persister(impl = EntityPersister.class)

public class Author {

@OneToMany(mappedBy = "author")

@Persister(impl = CollectionPersister.class)

public Set<Book> books = new HashSet<>();

# Access strategies

Hibernateподдерживает две стратеги доступа к атрибутам Entity:

1. Fieldbased (Прямой доступ к атрибуту через ReflectionAPI).
2. Property based (ДоступкатрибутамчерезихGetters and setters).

Если стратегия явно не указана через аннотацию @Access, то на будет неявно определяться по тому, где расположена аннотация @id- на атрибуте или свойстве (gettermethod).

Embeddable типы наследуют стратегию доступа от своих родительских entities (Но можно изменить).

Рекомендуется использовать field based strategy (<https://www.youtube.com/watch?v=3RGw86dQnBE>).

# Identifiers

Идентификаторы (identifiers) могут быть:

* simple (single value);
* composite (multiple values).

# Simple identifiers

СпецификацияJPAподдерживает следующие типы, которые могут выступать в качестве идентификаторов:

* any Java primitive type
* any primitive wrapper type
* java.lang.String
* java.util.Date (TemporalType#DATE)
* java.sql.Date
* java.math.BigDecimal
* java.math.BigInteger

# Compositeidentifiers

Требования спецификации Hibernateк составному идентификатору:

* Составной идентификатор должен быть представлен как "primarykeyclass" (PKC). Определить такой класс можно с помощью аннотаций:

1. javax.persistence.EmbeddedId (see [Composite identifiers with @EmbeddedId](https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#identifiers-composite-aggregated)).
2. javax.persistence.IdClass (see [Composite identifiers with @IdClass](https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#identifiers-composite-nonaggregated)).

* PKCдолжен быть**public**и иметь**public**конструктор без аргументов.
* PKCдолженбыть**serializable**.
* PKCдолженопределятьметоды**equals**и**hashCode**, consistent with equality for the underlying database types to which the primary key is mapped.

Атрибуты, входящие в состав PKC могут быть:

1. Basic.
2. Composite.
3. ManyToOne.

# Derivedidentifiers

Спецификация JPA 2.0 добавляетподдержкудля derived identifiers которыепозволяют an entity заимствовать identifier from a many-to-one или one-to-one association.

Сделать это можно через использование аннотаций:

* @MapsId;
* @PrimaryKeyJoinColumn

# @RowId

If you annotate a given entity with the @RowId annotation and the underlying database supports fetching a record by ROWID (e.g. Oracle), then Hibernate can use the ROWID pseudo-column for CRUD operations.

# Generated identifier values

Дляуказания того, что идентификатор будет генерироваться, используется аннотация javax.persistence.GeneratedValue. Наиболееважнаячастьинформацииуказываетсяв атрибуте strategy(возможные значения указываются с помощью enumjavax.persistence.GenerationType), которая определяет способ генерации идентификатора.

Перечисление javax.persistence.GenerationTypeвключает:

1. **AUTO (thedefault)**–указывает, чтоpersistenceprovider (Hibernate) долженвыбратьподходящуюстратегию.
2. **IDENTITY**–указывает, чтоdatabaseIDENTITYcolumnsбудутиспользованыдлягенерации значения первичного ключа.
3. **SEQUENCE**–указывает, чтоdatabasesequenceдолжнабытьиспользованадляполучения значения первичного ключа.
4. **TABLE**–указывает, что таблица базыданных должна быть использована для получения значения первичного ключа.

# Identity generation type

Hibernate will not be able to JDBC batching for inserts of the entities that use IDENTITY generation.

# Sequence generation type

Некоторые базы не поддерживают sequences(MySQL), но вместо этого они могут использовать таблицу в качестве sequence.

# Optimizers

Optimizersопределяют то, насколько часто Hibernateбудет обращаться к базе данных, для того чтобы сгенерировать необходимые идентификаторы.

Возможные optimizers:

* **NONE** - no optimization is performed.
* **POOLED-LO** - the pooled-lo optimizer works on the principle that the increment-value is encoded into the database table/sequence structure. In sequence-terms, this means that the sequence is defined with a greater-than-1 increment size.
* **POOLED** - just like pooled-lo, except that here the value from the table/sequence is interpreted as the high end of the value pool.
* **HILO; LEGACY-HILO** - Define a custom algorithm for generating pools of values based on a single value from a table or sequence. These optimizers are not recommended for use.

# @GenericGenerator

@GenericGenerator allows integration of any Hibernate org.hibernate.id.IdentifierGenerator implementation, including any of the specific ones discussed here and any custom ones.

@Id

@GeneratedValue(

strategy = GenerationType.SEQUENCE,

generator = "product\_generator"

)

@GenericGenerator(

name = "product\_generator",

strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",

parameters = {

@Parameter(name = "sequence\_name", value = "product\_sequence"),

@Parameter(name = "initial\_value", value = "1"),

@Parameter(name = "increment\_size", value = "3"),

@Parameter(name = "optimizer", value = "pooled-lo")

}

)

private Long id;

# GENERATEDPROPERTIES

**@Generated**

Генерируемые свойства (Generatedproperties)это свойства, значения которых сгенерированы базой данных. Свойства, помеченные как generatedдополнительно должнв бытьnon-insertableиnon-updateable. Только типы@Versionи@Basicмогут быть помечены какgenerated.

Чтобы пометить поле как генерируемое следует использовать специфическую Hibernateаннотацию@Generated(org.hibernate.annotations.Generated), которая принимает в качестве аргумента “Value” enum “GenerationTime”:

1. **GenerationTime.NEVER (the default)** -the given property value is not generated within the database.
2. **GenerationTime.INSERT** - the given property value is generated on insert but is not regenerated on subsequent updates. Properties like creationTimestamp fall into this category.
3. **GenerationTime.ALWAYS** - the property value is generated both on insert and update.

**@GeneratorType**:

The [@GeneratorType](https://docs.jboss.org/hibernate/orm/5.3/javadocs/org/hibernate/annotations/GeneratorType.html) annotation is used so that you can provide a custom generator to set the value of the currently annotated property.

**@CreationTimestamp:**

The @CreationTimestamp annotation instructs Hibernate to set the annotated entity attribute with the current timestamp value of the JVM when the entity is being persisted.

The supported property types are:

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

**@UpdateTimestamp:**

The @UpdateTimestamp annotation instructs Hibernate to set the annotated entity attribute with the current timestamp value of the JVM when the entity is being persisted.

The supported property types are:

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

**@ValueGenerationType** meta-annotation

Позволяет генерировать значение как на стороне базы данных, так и по аналогии с аннотацией **@GeneratorType**.

# COLUMN TRANSFORMERS: READ AND WRITE EXPRESSIONS

Позволяет модифицировать значение при его записи и чтении.

@Column(name = "pswd")

@ColumnTransformer(

read = "decrypt('AES', '00', pswd)",

write = "encrypt('AES', '00', ?)"

)

private String password;

You can use the plural form @ColumnTransformers if more than one columns need to define either of these rules.

# NAMINGSTRATEGIES

Двухэтапный маппинг:

1. Определение логического имени (Через явное указание имени или неявное определение через ImplicitNamingStrategy).
2. Определение физического имени на основании логического имени через PhysicalNamingStrategy.

В JPA-спецификации логическое имя и физическое имя – это одно и тоже. Соответственно, чтобы придерживаться JPA – в приложении не должна задаваться PhysicalNamingStrategy. Также для соответствия JPA необходимо придерживаться ImplicitNamingStrategyJpaCompliantImpl (the default) для определения неявных имен.

Когда в приложении явно не задается имя таблицы или атрибута (поля), то используется одна из реализаций ImplicitNamingStrategy.Hibernate имеет несколько готовых реализаций, но можно создавать и пользовательские.



Рисунок 6.1 Имплементации интерфейса ImplicitNamingStrategy

Способы задания ImplicitNamingStrategy:

1. Черезсвойствоконфигурацииhibernate.implicit\_naming\_strategy, которыйпринимает:

* **pre-defined "short names"** for the out-of-the-box implementations (“default”, “jpa”, “legacy-hbm”, “legacy-jpa”);
* **reference to a class** that implements the org.hibernate.boot.model.naming.ImplicitNamingStrategy contract;
* **FQN of a class** that implements the org.hibernate.boot.model.naming.ImplicitNamingStrategy contract.

1. Используя метод org.hibernate.boot.MetadataBuilder#applyImplicitNamingStrategy (uses by applications or integrations).

Способы задания PhysicalNamingStrategy:

1. Через свойство конфигурации hibernate.physical\_naming\_strategy, который принимает:

* **reference to a class** that implements the org.hibernate.boot.model.naming.PhysicalNamingStrategy contract;
* **FQN of a class** that implements the org.hibernate.boot.model.naming. PhysicalNamingStrategy contract.

1. Используя метод org.hibernate.boot.MetadataBuilder#applyPhysicalNamingStrategy (uses by applications or integrations)

Для конвертирования имен сущностей и их атрибутов в имена таблиц и их полей, которые должны соответсвовать определенной конвенции именования – используйте реализацию PhysicalNamingStrategy,так как она работает с логическими именами, которые могут быть заданы как явно, так и неявно.

# SQL QUOTEDIDENTIFIERS

Если в качестве параметре используется ключевое слово, то такие значения необходимо экранировать:

1. Hibernate: @Column(name=”`name`”).
2. JPA: @Column(name=”\”name\””).

Можно установить глобальное экранирование через свойство

<property

name="hibernate.globally\_quoted\_identifiers"

value="true"

/>

# ASSOCIATIONS

Четыре типа ассоциации:

1. ManyToOne.
2. OneToMany.
3. OneToOne.
4. ManyToMany.

# ManyToOne

**@ManyToOne** is the most common association, having a direct equivalent in the relational database as well (e.g. foreign key)

В БД реализуется через две таблицы (соединение по внешнему ключу).

В классах реализуется через аннотацию:

@JoinColumn(name = "person\_id",foreignKey = @ForeignKey(name = "PERSON\_ID\_FK")

в классе, предстовляющем множественную чторону отношения.

# OneToMany

Отношение OneToManyможет быть реализовано в двух вариантах:

1. **Unidirectional** – соответсвующие аннотации расположены только на стороне одиночного отношения.
2. **Bidirectional** – соответсвующие аннотации расположены на обеих сторонах отношения.

В бдUnidirectionalOneToManyотношение реализуется через дополнительную таблицу.

Параметры аннотации OneToMany:

1. **Cascade** – указывает, какие операции распространятся на другую сторону отношения.
2. **OrphanRemoval –** при значении **true**, если на стороне одиночного отношения удаляется ссылка на сущность со стороны множественного отношения, то эта сущность также будет удалена из БД, в противном случае будет удалена только само отношение сущностей.

При **unidirectionalOneToMany**отношении, когда на стороне одиночного отношения происходит удаление дочернего элемента из коллекции, то Hibernateделает это не эффективно – сначала удаляются все записи с ID родительского элемента, а потом вставляются все те, которые остались в коллекции.

**BiderectuioanlOneToMany**отношение требует наличие **ManyToOne**отношения на дочерней стороне. На стороне БД реализуется двумя таблицами (соединение по внешнему ключу).

Every bidirectional association must have one owning side only (the child side), the other one being referred to as the inverse (or the mappedBy) side.

Whenever a bidirectional association is formed, the application developer must make sure both sides are in-sync at all times (Черездополнительныеутилитныеметоды).

# OneToOne

Отношение OneToOneтакжеможет быть реализовано в двух вариантах:

1. **Unidirectional** – соответсвующие аннотации расположены только на стороне одиночного отношения.
2. **Bidirectional** – соответсвующие аннотации расположены на обеих сторонах отношения.

When using a bidirectional @OneToOne association, Hibernate enforces the unique constraint upon fetching the child-side. If there are more than one children associated with the same parent, Hibernate will throw aorg.hibernate.exception.ConstraintViolationException.

**Bidirectional @OneToOne lazy association**

Although you might annotate the parent-side association to be fetched lazily, Hibernate cannot honor this request since it cannot know whether the association is null or not.

The only way to figure out whether there is an associated record on the child side is to fetch the child association using a secondary query. Because this can lead to N+1 query issues, it’s much more efficient to use unidirectional @OneToOne associations with the @MapsId annotation in place.

However, if you really need to use a bidirectional association and want to make sure that this is always going to be fetched lazily, then you need to enable lazy state initialization bytecode enhancement and use the [@LazyToOne](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/LazyToOne.html) annotation as well.

# ManyToMany

Отношение ManyToManyтакжеможет быть реализовано в двух вариантах:

1. **Unidirectional** – соответсвующие аннотации расположены только на стороне одиночного отношения.
2. **Bidirectional** – соответсвующие аннотации расположены на обеих сторонах отношения.

The @ManyToMany association requires a link table that joins two entities.

В отличии от bidirectionalOneToMany, bidirectionalManyToManyне позволяет оптимизировать удаление дочерней записи из коллекции, содержащейся в родительское entity. Будет происходить удаление всех записей из промежуточной таблицы и повторная вставка всех записей, кроме удаленной. Для преодоления этого ограничения необходимо заменить отношение ManyToManyна два bidirectionalOneToMany.

# ManyToOne + Any

The @Any mapping is useful to emulate a unidirectional @ManyToOne association when there can be multiple target entities.Т.е. полеможетбытьзаданородительскимклассомилиинтерфейсом, ассылатьсянаегопотомков.

# ManyToAny

While the @Any mapping is useful to emulate a @ManyToOne association when there can be multiple target entities, to emulate a @OneToMany association, the @ManyToAny annotation must be used.

# ManyToOne + JoinFormula

The [@JoinFormula](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/JoinFormula.html) annotation is used to customize the join between a child Foreign Key and a parent row Primary Key.*The formula has to be a valid SQL fragment*.

# ManyToOne + JoinColumnOrFormula

The [@JoinColumnOrFormula](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/JoinColumnOrFormula.html) annotation is used to customize the join between a child Foreign Key and a parent row Primary Key when we need to take into consideration a column value as well as a @JoinFormula.

# COLLECTIONS

Persistent collection can contain:

* Basic types;
* Custom types;
* Embeddables;
* References to other entities.

Hibernate uses its own collection implementations, which are, enriched with lazy-loading, caching or state change detection semantics. For this reason, persistent collections must be declared as an interface type. The actual interface might be:

* java.util.Collection;
* java.util.List;
* java.util.Set;
* java.util.Map;
* java.util.SortedSet;
* java.util.SortedMap;
* implementation of org.hibernate.usertype.UserCollectionType.

The persistent collections injected by Hibernate behave like ArrayList, HashSet, TreeSet, HashMap or TreeMap, depending on the interface type.

Value and embeddable type collections have a similar behavior to basic types since they are automatically persisted when referenced by a persistent object and automatically deleted when unreferenced.

Two entities cannot share a reference to the same collection instance.

Collection-valued properties do not support null value semantics because Hibernate does not distinguish between a null collection reference and an empty collection.

# Collections of value types

Collections of value type include basic and embeddable types. Collections cannot be nested, and, when used in collections, embeddable types are not allowed to define other collections.

For collections of value types, JPA 2.0 defines the **@ElementCollection** annotation.

The lifecycle of the value-type collection is entirely controlled by its owning entity.

При удалении элемента из коллекции, происходит удаление всех записей из соответствующей таблицы и повторная вставка всех оставшихся. Одним из способов решения этой проблемы может быть аннотация **@OrderColumn**, которая добавляет дополнительное поле в таблицу коллекции.

The **@OrderColumn** column works best when removing from the tail of the collection, as it only requires a single delete statement. Removing from the head or the middle of the collection requires deleting the extra elements and updating the remaining ones to preserve element order.

# Collections of entities

Entity collections can be devised into two main categories:

* Unidirectional;
* Bidirectional.

Bidirectional associations are more tricky since, even if sides need to be in-sync at all times, only one side is responsible for managing the association.

A bidirectional association has:

* an *owning* side;
* an *inverse (mappedBy)* side.

Categorizing entity collections by the underlying collection type:

* bags;
* indexed lists;
* sets;
* sorted sets;
* maps;
* sorted maps;
* arrays.

# Bags

Bags are unordered lists, and we can have unidirectional bags or bidirectional ones.

**Unidirectional bags**

The unidirectional bag is mapped using a single **@OneToMany** annotation on the parent side of the association. Behind the scenes, Hibernate requires an association table to manage the parent-child relationship.

Just like value type collections, unidirectional bags are not as efficient when it comes to modifying the collection structure (removing or reshuffling elements).

Because the parent-side cannot uniquely identify each individual child, Hibernate deletes all link table rows associated with the parent entity and re-adds the remaining ones that are found in the current collection state.

**Bidirectional bags**

The bidirectional bag is the most common type of entity collection. The **@ManyToOne** side is the owning side of the bidirectional bag association, while the **@OneToMany** is the inverse side, being marked with the **mappedBy** attribute.

# Ordered lists

Bags don’t retain element order. To preserve the collection element order, there are two possibilities:

* **@OrderBy** - the collection is ordered upon retrieval using a child entity property;
* **@OrderColumn** - the collection uses a dedicated order column in the collection link table.

The **@OrderBy** annotation can take multiple entity properties, and each property can take an ordering direction too (e.g. @OrderBy("name ASC, type DESC")).

If no property is specified (e.g. @OrderBy), the primary key of the child entity table is used for ordering.

**@OrderColumn** - You can customize the ordinal of the underlying ordered list by using the [**@ListIndexBase**](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/ListIndexBase.html) annotation.

**@OrderBy -** While the JPA [@OrderBy](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/OrderBy.html) annotation allows you to specify the entity attributes used for sorting when fetching the current annotated collection, the Hibernate specific [@OrderBy](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/OrderBy.html) annotation is used to specify a **SQL** clause instead.

# Sets

Hibernate supports both the unordered Set and the natural-ordering SortedSet.

When using Sets, it’s very important to supply proper equals/hashCode implementations for child entities.

In the absence of a custom equals/hashCode implementation logic, Hibernate will use the default Java reference-based object equality which might render unexpected results when mixing detached and managed object instances.

**SortedSets:**

* **@SortComparator** specifies comparator class which implements **Comparator** interface;
* **@SortNatural** uses sorting with **Comparable** interface implemented by entity.

**@SortComparator** and **@SortNatural** performs in-memory sorting for Set and Map. Their functions are different from**@OrderBy** because it is applied during SQL SELECT.

# Maps

Hibernate allows using the following map keys:

* **MapKeyColumn** - for value type maps, the map key is a column in the link table that defines the grouping logic;
* **MapKey** - the map key is either the primary key or another property of the entity stored as a map entry value;
* **MapKeyEnumerated** - the map key is an Enum of the target child entity;
* **MapKeyTemporal** - the map key is a Date or a Calendar of the target child entity;
* **MapKeyJoinColumn** - the map key is an entity mapped as an association in the child entity that’s stored as a map entry key.

Additional information for understanding maps keys - <https://www.baeldung.com/hibernate-persisting-maps>

При рассмотрении отношения между Entityи содержащейся в ней мапе следуют понимать это отношение, как отношение Entity кзначению мапы. Если значание имеет value тип, то используем **@ElementCollection**, если Entityтип, то One/ManyToManyаннотацию (Моезамечание).

A map of value type must use the **@ElementCollection** annotation.

Entity with a map of entity type can have unidirectional and bidirectional associations.В качестве ключа карты может выступать Interface.

# Arrays

Hibernate does support the mapping of arrays in the Java domain model - conceptually the same **as mapping a List**. However, it is important to realize that it is impossible for Hibernate to offer lazy-loading for arrays of entities and, for this reason, it is strongly recommended to map a "collection" of entities using a List rather than an array.

Additional information - <https://thorben-janssen.com/mapping-arrays-with-hibernate/>

You can persist array as binary.

# Collection as basic value type

Collections not marked as @**ElementCollection**, **OneToMany**, **ManyTOMany**require a custom Hibernate Type and the collection elements must be stored in a single database column.

# Custom collection type

You can implement your own type of collections. Queue for example.

# NATURAL IDS

В качестве NaturalIDможно использовать:

1. Одиночное поле Entity.
2. Embeddable class.
3. Составное поле.

ПризагрузкеentityпоnaturalIDформируется два запроса, сначала находится IDкласса по NaturalID, потом сам класс по найденному ID. Однако если будут созданы необходимые индексы, проблем с производительностью не будет. К тому же для naturalIDможно использовать Cahing (@NaturalIdCache для entity).

Для получения entity с помощью класса NaturalIdLoadAccess существует два метода:

* **load()** - obtains a reference to the entity, making sure that the entity state is initialized.
* **getReference() -** obtains a reference to the entity. The state may or may not be initialized. If the entity is already associated with the current running Session, that reference (loaded or not) is returned. If the entity is not loaded in the current Session and the entity supports proxy generation, an uninitialized proxy is generated and returned, otherwise the entity is loaded from the database and returned.

Для обозначения naturalIdиспользуется аннотация @NaturalId. Данная аннотация имеет параметр **mutable**, со значение по умолчанию – false (Можно настроить оптимизацию исполнения).

# DYNAMIC MODEL

Hibernate позволяет сохранять объект в базу, имея лишь его описание в конфигурационном файле, наличие самого класса не требуется. Может применяться для прототипирования.

# INHERITANCE

Although relational database systems don’t provide support for inheritance, Hibernate provides several strategies to leverage this object-oriented trait onto domain model entities:

**MappedSuperclass** - Inheritance is implemented in the domain model only without reflecting it in the database schema.

Because the @MappedSuperclass inheritance model is not mirrored at the database level, it’s not possible to use polymorphic queries referencing the @MappedSuperclass when fetching persistent objects by their base class.

**Singletable** - The domain model class hierarchy is materialized into a single table which contains entities belonging to different class types.

When omitting an explicit inheritance strategy (e.g. @Inheritance), JPA will choose the SINGLE\_TABLE strategy by default.

Version and id properties are assumed to be inherited from the root class.

**Discriminator**

Each subclass in a hierarchy must define a unique discriminator value, which is used to differentiate between rows belonging to separate subclass types. If this is not specified, the DTYPE column is used as a discriminator, storing the associated subclass name.

The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row. Hibernate Core supports the following restricted set of types as discriminator column: String, char, int, byte, short, boolean(including yes\_no, true\_false).

Among all other inheritance alternatives, the single table strategy performs the best since it requires access to one table only. Because all subclass columns are stored in a single table, it’s not possible to use NOT NULL constraints anymore, so integrity checks must be moved either into the data access layer or enforced through CHECK or TRIGGER constraints.

**Joinedtable** - The base class and all the subclasses have their own database tables and fetching a subclass entity requires a join with the parent table as well.

**Table per class** - Each subclass has its own table containing both the subclass and the base class properties.A requirement for all child objects of the same parent entity is that they have unique IDs among them(?).

**Polymorphic Queries**

By default, when you query a base class entity, the polymorphic query will fetch all subclasses belonging to the base type.

However, you can even query **interfaces or base classes that don’t belong to the JPA entity inheritance model**.

When we query against the superclass orinterface, Hibernate is going to fetch only the entities that are either mapped with @Polymorphism(type = PolymorphismType.IMPLICIT) or they are not annotated at all with the @Polymorphism annotation (implying the IMPLICIT behavior).

# IMMUTABILITY

Хорошей практикой является пометка неизменяемых Entities и Collections как **@Immutable**. Это позволяет Hibernate проводить определенные оптимизации.

При попытке изменить:

* **Immutable entity** - запрос на обновление будет проигнорирован;
* **Immutable collection** - будет выброшено исключение.

# BOOTSTRAP

We can interfere with the process of bootstrapping via:

1. Native bootstrapping.
2. JPA bootstrapping.

# SCHEMA GENERATION

Hibernate allows you to generate the database from the entity mappings.

* To customize the schema generation process, the hibernate.hbm2ddl.import\_files configuration property must be used to provide other scripts files that Hibernate can use when the SessionFactory is started.

<property name="hibernate.hbm2ddl.import\_files"

value="schema-generation.sql" />

* Hibernate allows you to customize the schema generation process via the HBM database-object element.

<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd" >

<hibernate-mapping>

**<database-object>**

**<create>**

**CREATE OR REPLACE FUNCTION sp\_count\_books(**

**IN authorId bigint,**

**OUT bookCount bigint)**

**RETURNS bigint AS**

**$BODY$**

**BEGIN**

**SELECT COUNT(\*) INTO bookCount**

**FROM book**

**WHERE author\_id = authorId;**

**END;**

**$BODY$**

**LANGUAGE plpgsql;**

**</create>**

**<drop></drop>**

**<dialect-scope name="org.hibernate.dialect.PostgreSQL95Dialect" />**

**</database-object>**

</hibernate-mapping>

* Hibernate offers the **@Check** annotation so that you can specify an arbitrary SQL CHECK constraint which can be defined as follows:

@Entity(name = "Book")

**@Check(constraints = "CASE WHEN isbn IS NOT NULL THEN LENGTH(isbn) = 13 ELSE true END")**

public static class Book {

* With Hibernate, you can specify a default value for a given database column using the [**@ColumnDefault**](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/ColumnDefault.html) annotation;

**@ColumnDefault("'N/A'")**

private String name;

* The [**@UniqueConstraint**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/UniqueConstraint.html) annotation is used to specify a unique constraint to be included by the automated schema generator for the primary or secondary table associated with the current annotated entity;

@Entity

@Table(

name = "book",

**uniqueConstraints = @UniqueConstraint(**

**name = "uk\_book\_title\_author",**

**columnNames = {**

**"title",**

**"author\_id"**

**}**

**)**

)

public static class Book {

* The [**@Index**](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Index.html) annotation is used by the automated schema generation tool to create a database index.

@Entity

@Table(

name = "author",

**indexes = @Index(**

**name = "idx\_author\_first\_last\_name",**

**columnList = "first\_name, last\_name",**

**unique = false**

**)**

)

public static class Author {

# PERSISTENCE CONTEXT

Persistent data has states:

**Transient -** the entity has just been instantiated and is not associated with a persistence context. It has no persistent representation in the database and typically no identifier value has been assigned (unless the *assigned* generator was used).

**Mmanaged or persistent -** the entity has an associated identifier and is associated with a persistence context. It may or may not physically exist in the database yet.

**Detached -** the entity has an associated identifier but is no longer associated with a persistence context (usually because the persistence context was closed or the instance was evicted from the context)

**Removed -** the entity has an associated identifier and is associated with a persistence context, however, it is scheduled for removal from the database.

**Accessing Hibernate APIs from JPA:**

Session session = entityManager.unwrap( Session.class );

SessionImplementor sessionImplementor = entityManager.unwrap( SessionImplementor.class );

SessionFactory sessionFactory = entityManager.getEntityManagerFactory().unwrap( SessionFactory.class );

# BYTECODE ENHANCEMENT

Hiberante включает в себя механизм “BYTECODE ENHANCEMENT”, который позволяет вносить изменения в байт код классов. Так, например, при стандартной проверке на изменение аттрибутов entitites используется механизм “Dirty checking”. При загрузке entity в контексте происходит запоминание состояния, а при выполнении операции flush() производится сравнение текущего состояния с сохраненным. Но при такой схеме приходится сравнивать и хранить состояния для всех entities контекста. Если же использовать “Bytecode enhancement”, то при компиляции entities происходит добавление в классы дополнительного кода, который сам будет отслеживать изменения в сущности и, при необходимости, сообщать об этом.

Hibernate supports the enhancement of an application Java domain model for the purpose of adding various persistence-related capabilities directly into the class:

1. Lazy attribute loading (Позволяет выполнять lazy loading даже для базовых типов (не точно, надо уточнять)).
2. In-line dirty tracking.
3. Bidirectional association management (При инициализации этого свойства, будет проводится автоматическая синхронизация связанных сущностей при установке свойства только для одной из entitites, helper методы не нужны).

To enable runtime enhancement, specify one of the following configuration properties:

**hibernate.enhancer.enableDirtyTracking** (e.g. true or false (default value)) -

Enable dirty tracking feature in runtime bytecode enhancement.

**hibernate.enhancer.enableLazyInitialization** (e.g. true or false (default value))

Enable lazy loading feature in runtime bytecode enhancement. This way, even basic types (e.g. @Basic(fetch = FetchType.LAZY)) can be fetched lazily.

**hibernate.enhancer.enableAssociationManagement** (e.g. true or false (default value))

Enable association management feature in runtime bytecode enhancement which automatically synchronizes a bidirectional association when only one side is changed.

Hibernate provides a Maven plugin capable of providing build-time enhancement of the domain model as they are compiled as part of a Maven build (**hibernate-enhance-maven-plugin**).

# MAKING ENTITIES PERSISTENT

Once you’ve created a new entity instance (using the standard new operator) it is in new state. You can make it persistent by associating it to either an org.hibernate.Session or a javax.persistence.EntityManager:

1. EntityManager.persist().
2. Session.save().

# DELETING ENTITIES

Entities can also be deleted:

1. EntityManager.remove().
2. Session.delete().

Hibernate itself can handle deleting entities in detached state. JPA, however, disallows this behavior (?).

# OBTAINING AN ENTITY REFERENCE WITHOUT INITIALIZING ITS DATA

Sometimes referred to as lazy loading, the ability to obtain a reference to an entity without having to load its data is hugely important. The most common case being the need to create an association between an entity and another existing entity:

1. EntityManager.getReference().
2. Session.load().

Unless the entity class is declared final, the proxy extends the entity class. If the entity class is final, the proxy will implement an interface instead.

# OBTAINING AN ENTITY REFERENCE WITH INITIALIZING ITS DATA

It is also quite common to want to obtain an entity along with its data:

1. EntityManager.find().
2. Session.get(), session.byId(Some.class).load(someId), session.byId(Some.class) .loadOptional(someId).

# OBTAIN MULTIPLE ENTITIES BY THEIR IDENTIFIERS

(Hibernate only!) While the JPA standard does not support retrieving multiple entities at once, other than running a JPQL or Criteria API query, Hibernate offers this functionality via the [**byMultipleIds** method](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/Session.html#byMultipleIds-java.lang.Class-) of the Hibernate Session.

# FILTERING ENTITIES AND ASSOCIATIONS

Hibernate offers two options if you want to filter entities or entity associations:

* **Static** (**@Where**, **@WhereJoinTable**) - which are defined at mapping time and cannot change at runtime.
* **Dynamic** (**@Filter**, **@FilterJoinTable**) - which are applied and configured at runtime.

При статической фильтрации условие фильтра задается на этапе комиляции, при динамической – в рантайме можно задавать значения параметров фильтров.

**@WhereJoinTable** и **@FilterJoinTable** используются в случаях, когда для организации связи используется join table.

# MODIFYING MANAGED/PERSISTENT STATE

Entities in managed/persistent state may be manipulated by the application, and any changes will be automatically detected and persisted when the persistence context is flushed. There is no need to call a particular method to make your modifications persistent.

By default, when you modify an entity, all columns but the identifier are being set during update. The default UPDATE statement containing all columns has two advantages:

* It allows you to better benefit from JDBC Statement caching.
* It allows you to enable batch updates even if multiple entities modify different properties.

The dynamic update allows you to set just the columns that were modified in the associated entity. To enable dynamic updates, you need to annotate the entity with the **@DynamicUpdate** annotation.

# REFRESH ENTITY STATE

**Refresh()** method allow to refresh entity state.

# ССЫЛКИ

1. Материал для повторения! – <https://habr.com/ru/post/265061/>
2. Difference between first level and second level cache in Hibernate. - <https://www.java67.com/2017/10/difference-between-first-level-and-second-level-cache-in-Hibernate.html>
3. Cool man’s tutorial - <https://vladmihalcea.com/tutorials/hibernate/>
4. [@NaturalId – A good way to persist natural IDs with Hibernate?](https://thorben-janssen.com/naturalid-good-way-persist-natural-ids-hibernate/) - <https://thorben-janssen.com/naturalid-good-way-persist-natural-ids-hibernate/>
5. Oracle docs (Developers Guide for JPA/JDO) - <https://docs.oracle.com/html/E13946_04/>.
6. Cashing - <https://habr.com/ru/post/135176/>.

<https://www.red-gate.com/simple-talk/sql/database-administration/using-migration-scripts-in-database-deployments/>

https://thorben-janssen.com/hibernate-tips-remove-entities-persistence-context/