# Оглавление

[Оглавление 1](#_Toc74054007)

[1. HIBERNATE 4](#_Toc74054008)

[2. MAPPING TYPES 5](#_Toc74054009)

[2.1. VALUE TYPES (BASIC) 5](#_Toc74054010)

[2.1.1. MAPPING ENUMS 8](#_Toc74054011)

[2.1.2. ATTRIBUTE CONVERTOR 8](#_Toc74054012)

[2.1.3. MAPPING LARGE OBJECTS (LOB) 8](#_Toc74054013)

[2.1.4. MAPPING NATIONALIZED CHARACTER DATA 9](#_Toc74054014)

[2.1.5. MAPPING UUID VALUES 9](#_Toc74054015)

[2.1.6. MAPPING DATA/TIME VALUES 9](#_Toc74054016)

[2.2. VALUE TYPES (EMBEDDABLE) 11](#_Toc74054017)

[3. ENTITY TYPES 11](#_Toc74054018)

[3.1. Implementing equals() and hashCode() 12](#_Toc74054019)

[3.2. Mapping the entity to a SQL query 13](#_Toc74054020)

[3.3. Definea custom entity proxy 13](#_Toc74054021)

[3.4. Define a custom entity persister 13](#_Toc74054022)

[3.5. Access strategies 13](#_Toc74054023)

[3.6. Identifiers 13](#_Toc74054024)

[3.6.1. Simple identifiers 14](#_Toc74054025)

[3.6.2. Compositeidentifiers 14](#_Toc74054026)

[3.6.3. Derivedidentifiers 14](#_Toc74054027)

[3.6.4. @RowId 14](#_Toc74054028)

[3.7. Generated identifier values 15](#_Toc74054029)

[3.7.1. Identity generation type 15](#_Toc74054030)

[3.7.2. Sequence generation type 15](#_Toc74054031)

[3.8. Optimizers 15](#_Toc74054032)

[3.9. @GenericGenerator 16](#_Toc74054033)

[4. GENERATEDPROPERTIES 16](#_Toc74054034)

[5. COLUMN TRANSFORMERS: READ AND WRITE EXPRESSIONS 17](#_Toc74054035)

[6. NAMINGSTRATEGIES 17](#_Toc74054036)

[7. SQL QUOTEDIDENTIFIERS 19](#_Toc74054037)

[8. ASSOCIATIONS 19](#_Toc74054038)

[8.1. ManyToOne 19](#_Toc74054039)

[8.2. OneToMany 19](#_Toc74054040)

[8.3. OneToOne 20](#_Toc74054041)

[8.4. ManyToMany 20](#_Toc74054042)

[8.5. ManyToOne + Any 21](#_Toc74054043)

[8.6. ManyToAny 21](#_Toc74054044)

[8.7. ManyToOne + JoinFormula 21](#_Toc74054045)

[8.8. ManyToOne + JoinColumnOrFormula 21](#_Toc74054046)

[9. COLLECTIONS 21](#_Toc74054047)

[9.1. Collections of value types 22](#_Toc74054048)

[9.2. Collections of entities 22](#_Toc74054049)

[9.2.1. Bags 23](#_Toc74054050)

[9.2.2. Ordered lists 23](#_Toc74054051)

[9.2.3. Sets 24](#_Toc74054052)

[9.2.4. Maps 24](#_Toc74054053)

[9.2.5. Arrays 24](#_Toc74054054)

[9.2.6. Collection as basic value type 25](#_Toc74054055)

[9.2.7. Custom collection type 25](#_Toc74054056)

[10. NATURAL IDS 25](#_Toc74054057)

[11. DYNAMIC MODEL 25](#_Toc74054058)

[12. INHERITANCE 26](#_Toc74054059)

[13. IMMUTABILITY 26](#_Toc74054060)

[14. BOOTSTRAP 27](#_Toc74054061)

[15. SCHEMA GENERATION 27](#_Toc74054062)

[16. PERSISTENCE CONTEXT 28](#_Toc74054063)

[16.1. BYTECODE ENHANCEMENT 29](#_Toc74054064)

[16.2. MAKING ENTITIES PERSISTENT 29](#_Toc74054065)

[16.3. DELETINGENTITIES 30](#_Toc74054066)

[16.4. OBTAINING AN ENTITY REFERENCE WITHOUT INITIALIZING ITS DATA 30](#_Toc74054067)

[16.5. OBTAINING AN ENTITY REFERENCE WITH INITIALIZING ITS DATA 30](#_Toc74054068)

[16.6. OBTAIN MULTIPLE ENTITIES BY THEIR IDENTIFIERS 30](#_Toc74054069)

[16.7. FILTERING ENTITIES AND ASSOCIATIONS 30](#_Toc74054070)

[16.8. MODIFYINGMANAGED/PERSISTENTSTATE 31](#_Toc74054071)

[16.9. REFRESH ENTITY STATE 31](#_Toc74054072)

[16.10. WORKING WITH DETACHED DATA 31](#_Toc74054073)

[16.10.1. Reattaching detached data 31](#_Toc74054074)

[16.10.2. Merging detached data 31](#_Toc74054075)

[16.11. CHECKING PERSISTENT STATE 32](#_Toc74054076)

[16.12. EVICTING ENTITIES 33](#_Toc74054077)

[17. CASCADING ENTITY STATE TRANSITIONS 33](#_Toc74054078)

[17.1. @OnDelete CASCADE 34](#_Toc74054079)

[18. FLUSHING 34](#_Toc74054080)

[19. DATABASE ACCESS 34](#_Toc74054081)

[19.1. ConnectionProvider support for transaction isolation setting 35](#_Toc74054082)

[19.2. Connection handling 36](#_Toc74054083)

[19.3. Transaction type and connection handling 37](#_Toc74054084)

[20. LOCKING 37](#_Toc74054085)

[20.1. Optimistic locking (OL) 38](#_Toc74054086)

[20.2. Pessimistic locking (PL) 39](#_Toc74054087)

[20.3. JPA locking query hints 40](#_Toc74054088)

[21. FETCHING 41](#_Toc74054089)

[21.1. FETCH TYPES 41](#_Toc74054090)

[21.2. Dynamic fetching via JPA entity graph 42](#_Toc74054091)

[21.3. Dynamic fetching via Hibernate profiles 42](#_Toc74054092)

[21.4. Static fetching via Hibernate specific @Fetch annotation 42](#_Toc74054093)

[21.5. Static fetching via Hibernate specific @LazyCollection anotaion 43](#_Toc74054094)

[22. BATCHING 43](#_Toc74054095)

[22.1. Session scroll 43](#_Toc74054096)

[22.2. StatelessSession 44](#_Toc74054097)

[23. CACHING 45](#_Toc74054098)

[23.1. Configuring second-level caching 46](#_Toc74054099)

[23.2. Configuring second-level cache mappings 47](#_Toc74054100)

[23.3. Entity inheritance and second-level cache mapping 48](#_Toc74054101)

[23.4. Entity cache 48](#_Toc74054102)

[23.5. Collection cache 48](#_Toc74054103)

[23.6. Query cache 48](#_Toc74054104)

[23.7. Managing the cached data 48](#_Toc74054105)

[23.8. Caching statistics 49](#_Toc74054106)

[23.9. Doc references 49](#_Toc74054107)

[24. INTERCEPTORS AND EVENTS 49](#_Toc74054108)

[24.1. Interceptors 49](#_Toc74054109)

[24.2. Native event system 50](#_Toc74054110)

[24.3. JPA callbacks 50](#_Toc74054111)

[24.4. Default entity listeners 50](#_Toc74054112)

[25. HQL, JPQL 51](#_Toc74054113)

[25.1. JPA query API 51](#_Toc74054114)

[25.2. Hibernate query API 52](#_Toc74054115)

[25.3. Query scrolling 52](#_Toc74054116)

[25.4. Query streaming 52](#_Toc74054117)

[25.5. Case sensitivity 52](#_Toc74054118)

[25.6. Statement types 53](#_Toc74054119)

[25.6.1. Select 53](#_Toc74054120)

[25.6.2. Update 53](#_Toc74054121)

[25.6.3. Delete 53](#_Toc74054122)

[25.6.4. Insert 54](#_Toc74054123)

[25.7. Distinct 54](#_Toc74054124)

[26. CRITERIA API 54](#_Toc74054125)

[27. Native queries 54](#_Toc74054126)

[28. ССЫЛКИ 55](#_Toc74054127)

# HIBERNATE

**Hibernate** – JPA-провайдер, который реализует (JavaPersistenceAPI) и расширяет! (Hibernate Native API - HNA) интерфейс JPA.



Рисунок1.1Структура приложения, использующего Hibernate.

**Примечание**: длятого, 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.

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

SessionFactory поддерживает сервисы, которые Hibernate используется для всех Sessions:

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

При создании EntityManagerFactory с автогенерацией DB (*hibernate.hbm2ddl.auto = update or create or …*) генерация происходит в момент создания EntityManagerFactory.

**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 underlying transaction system in use (JDBCorJTA - <https://laliluna.com/jpa-hibernate-guide/ch13s02.html>).

# MAPPING TYPES

**Hibernate type** не является ни Java типом, ни SQL типом. Hibernatetype предоставляет информацию о том, как отобразить Javatype на SQL type, и как 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).

Value types 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**  **(generic SQL type)** | **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 @Basic annotation on a field or a property signifies that it's a basic type and Hibernate should use the standard mapping for its persistence.**

**За выбор подходящего Hibernate-типа отвечает** org.hibernate.type.BasicTypeRegistry**, который позволяет 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).

Атрибуты аннотации @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 hibernate тип 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>).

**Примечание**: Mappings of JDBC Types to Database Vendor Specific Types - <https://docs.oracle.com/cd/E19830-01/819-4721/beajw/index.html> (table 7-2)

**Примечание** (не точно): Если один и тот же Java тип может мапиться на несколько JDBC (стандартных sql типов), то без явного указания hibernate типа, этот тип будет выбран автоматически. Чтобы этого избежать необходимо указать тип явно. Можно создавать свои базовые типы, и переопределять существующие.

**Hibernate dialect** отвечает за правильное преобразование данных из JDBC в vendor specific.

# MAPPING ENUMS

Hibernate позволяет отображать Enum в качестве базового типа. Для этого используется аннотация **@Enumerated** с параметром:

* **EnumType.ORDINAL** (поле будет сохранено как число);
* **EnumType.STRING** (поле будет сохранено как строка).

Можно также замапить Enum с помощью пользовательского типа (custom types).

# ATTRIBUTE CONVERTOR

Attribute conventer позволяет преобразовывать сохраняемое/извлекаемое значение. Например, если у нас есть enum Gender (MALE, FEMALE), то в базу можно сохранять “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 used as a query parameter (<https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#basic-enums>).

AttributeConverter может быть использован и при HBM маппинге.

# MAPPING LARGE OBJECTS (LOB)

Для мапинга LOB можно использовать:

1. JDBC locator types.
2. Types that 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. Для этого используется аннотация **@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 использовать:

1. Классы даты и времени из пакета java.sql.\*.
2. Классы даты и времени из пакета java.util.\* (Date и Calendar).
3. Java 8 Data/Time.

SQL стандарт определяет три типа Date/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** не имеют однозначного соответствия sql-типам базы даных, а поэтому должны специфициоваться дополнительной аннотацией **@Temporal** с указанием типа в параметре:

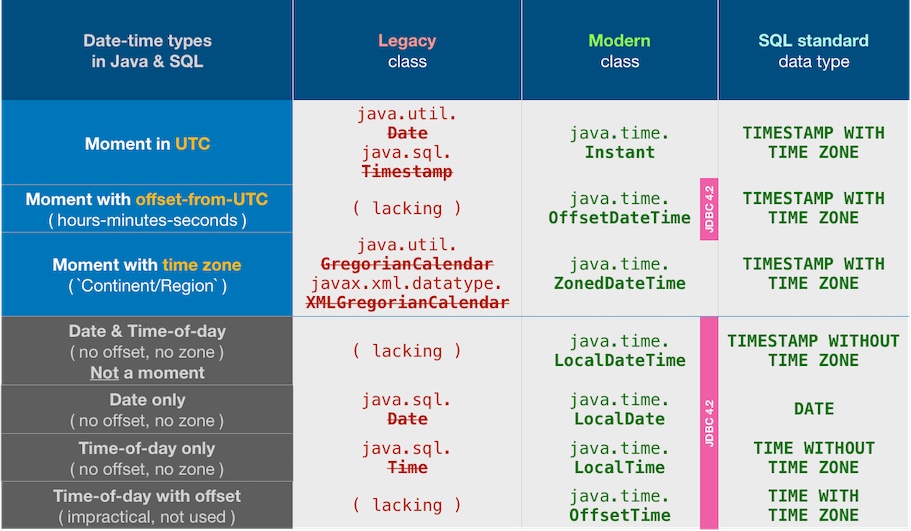
* **TemporalType.DATE;**
* **TemporalType.TIME;**
* **TemporalType.TIMESTAMP.**

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.

<https://stackoverflow.com/questions/2305973/java-util-date-vs-java-sql-date> :



**Использование специфической временной зоны:**

When the time zone is not specified, the JDBC driver is going to use the underlying JVM default time zone, which might not be suitable if the application is used from all across the globe. For this reason, it is very common to use a single reference time zone (e.g. UTC) whenever saving/loading data from the database.

One alternative would be to configure all JVMs to use the reference time zone:

Declaratively

java -Duser.timezone=UTC ...

Programmatically

TimeZone.setDefault(TimeZone.getTimeZone("UTC"));

However, as explained in [this article](https://in.relation.to/2016/09/12/jdbc-time-zone-configuration-property/), this is not always practical, especially for front-end nodes. For this reason, Hibernate offers the **hibernate.jdbc.time\_zone** configuration property:

Declaratively, at the SessionFactory level

settings.put(

AvailableSettings.JDBC\_TIME\_ZONE,

TimeZone.getTimeZone( "UTC" )

);

Programmatically, on a per Session basis

Session session = sessionFactory()

.withOptions()

.jdbcTimeZone( TimeZone.getTimeZone( "UTC" ) )

.openSession();

# VALUE TYPES (EMBEDDABLE)

Embeddable types или components поддерживают концепцию составных значений. В java модели 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).

Embeddable types that are used as

* Collection entries;
* Map keys;
* Entity type identifiers.

cannot include their own collection mappings.

Если Embeddable классы реализуют интерфейс, то в этом случае могут применяться аннотации:

* **@Target -** is used to define a custom join association between the parent-child association;
* **@Parent -** is used to define the association between an embeddable type and the owning entity.

Для аннотаций:

* **@ManyToOne;**
* **@OneToOne;**
* **@OneToMany;**
* **@ManyToMany;**
* **@ElementCollecion.**

Существует атрибут, подобный аннотации @Target, который позволяет указать класс, реализующий интерфейс.

# ENTITY TYPES

Требования JPA-спецификации к entity class:

1. Класс должен быть обозначен аннотацией @javax.persistnce.Entity или описан в xml-файле.
2. Класс должен иметь public или protected no-argument конструктор. Дополнительные конструкторы разрешены.
3. Класс должен быть top-level class.
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. Entity класс может иметь no-argument constructor, который может иметь public, protected или package видимость. Допускаются дополнительные конструкторы.
2. Entity класс не обязательно должен быть top-level класс.
3. Технически, Hibernate может persist final классы или классы с final 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 @Entity определяет только один аттрибут name, который дает конкретной Entity имя, которое будет использовано в JPQLзапросах. Поумолчанию, имя entity представляет из себя имя самого класса entity.

An entity models a data base table. 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 entity класс 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 Entities и 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

Hibernate поддерживает две стратеги доступа к атрибутам Entity:

1. Field based (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 родительскихentities (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

Спецификация JPA поддерживает 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

# Composit eidentifiers

Требования спецификации Hibernate к составному идентификатору:

* Составной идентификатор должен быть представлен как "primary key class" (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)).

* PKC должен быть **public** и иметь **public** конструктор без аргументов.
* PKC должен быть **serializable**.
* PKC должен определять методы **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 добавляет поддержку для derived identifiers, которые позволяют 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

Для указания того, что идентификатор будет генерироваться, используется аннотация **javax.persistence.GeneratedValue**. Наиболее важная часть информации указывается в атрибуте **strategy** (возможные значения указываются с помощью enum**javax.persistence.GenerationType**), которая определяет способ генерации идентификатора.

Перечисление **javax.persistence.GenerationType** включает:

1. **AUTO (thedefault)** – указывает, что persistence provider (Hibernate) должен выбрать подходящую стратегию.
2. **IDENTITY** – указывает, что database IDENTITY columns будут использованы для генерации значения первичного ключа.
3. **SEQUENCE** – указывает, что database sequence должна быть использована для получения 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

Optimizers определяют то, насколько часто Hibernate будет 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;

# GENERATED PROPERTIES

**@Generated**

Генерируемые свойства (Generated properties) это свойства, значения которых сгенерированы базой данных. Свойства, помеченные как generated дополнительно должны быть non-insertable и non-updateable. Только типы@Version и @Basic могут быть помечены как generated.

Чтобы пометить поле как генерируемое следует использовать специфическую Hibernate аннотацию @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.

ColumnTransformer us AttributeConventor - <https://stackoverflow.com/questions/49046265/difference-between-jpas-attributeconverter-and-hibernates-columntransformer-in>

# NAMING STRATEGIES

В Hibernate используется 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 и физическое имя – это одно и тоже. Соответственно, чтобы придерживаться JPA – в приложении не должна задаваться PhysicalNamingStrategy. Также для соответствия JPA необходимо придерживаться ImplicitNamingStrategyJpaCompliantImpl (thedefault) 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. Через свойство конфигурации hibernate.implicit\_naming\_strategy, который принимает:

* **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)

Описанное разделение стратегий именования необходимо для большей гибкости: на первом этапе определяются логические имена (то как имена классов и их полей будут отражены на имена таблиц и колонок в них). При этом не важно, как эти имена будут заданы – явно или неявно (за неявное именование, когда пользователь не задает имя явно, отвечает ImplicitNamingStrategy).

На втором же этапе, физические имена, которые будут отражены в бд, можно оставить эквивалентными логическим, либо провести их преобразование (например, полные наименования заменять абревеатурами).

# SQL QUOTED IDENTIFIERS

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 сторону отношения.

# OneToMany

Отношение OneToMany может 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 бд Unidirectional OneToMany отношение 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 **unidirectional OneToMany** 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.

**Biderectuioanl OneToMany** 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 (Через дополнительные утилитные методы).

# OneToOne

Отношение OneToOne также может 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 ManyToMany также может 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 bidirectional OneToMany, bidirectional ManyToManyне 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 bidirectional OneToMany.

# ManyToOne + Any

The @Any mapping is useful to emulate a unidirectional @ManyToOne association when there can be multiple target entities.Т.е. поле может быть задано родительским классом или интерфейсом, а ссылаться на его потомков.

# 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;
* 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;
* 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;
* 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.

**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.Вкачествеключакартыможетвыступать 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 NaturalID можно использовать:

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

При загрузке entity по naturalID формируется два запроса, сначала находится ID класса по NaturalID, потом сам класс по найденному ID. Однако если будут созданы необходимые индексы, проблем с производительностью не будет. К тому же для natural ID можно 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 naturalId используется 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 or interface, 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.

<propertyname="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 entityв 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 “Byte code 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 (неточно, 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:

Book book = new Book();

book.setAuthor( entityManager.getReference( Person.class, personId ) );

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 jointable.

# 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.

One case where this is useful is when it is known that the database state has changed since the data was read. Refreshing allows the current database state to be pulled into the entity instance and the persistence context.

Another case where this might be useful is when database triggers are used to initialize some of the properties of the entity.

Only the entity instance and its value type collections are refreshed unless you specify REFRESH as a cascade style of any associations. However, please note that Hibernate has the capability to handle this automatically through its notion of generated properties.

# WORKING WITH DETACHED DATA

# Reattaching detached data

**Reattachment** is the process of taking an incoming entity instance that is in the detached state and re-associating it with the current persistence context. JPA does not support reattaching detached data.

* Reattaching a detached entity using lock – **session.lock(person, LockMode.NONE)**;
* Reattaching a detached entity using saveOrUpdate - **session.saveOrUpdate(person)**;

# Merging detached data

**Merging** is the process of taking an incoming entity instance that is in the detached state and copying its data over onto a new managed instance.

Although not exactly per se, the following example is a good visualization of the merge operation internals.

public Person merge(Person detached) {

Person newReference = session.byId(Person.class)

.load(detached.getId());

newReference.setName(detached.getName());

return newReference;

}

Hibernate throws IllegalStateException when merging a parent entity which has references to 2 detached child entities child1 and child2 (obtained from different sessions), and child1 and child2 represent the same persistent entity, Child.

A new configuration property, **hibernate.event.merge.entity\_copy\_observer**, controls how Hibernate will respond when multiple representations of the same persistent entity ("entity copy") is detected while merging.

The possible values are:

* **disallow (the default)** - throws IllegalStateException if an entity copy is detected
* **allow** - performs the merge operation on each entity copy that is detected
* **log** - (provided for testing only) performs the merge operation on each entity copy that is detected and logs information about the entity copies. This setting requires DEBUG logging be enabled for org.hibernate.event.internal.EntityCopyAllowedLoggedObserver
* **custom implementation …**

**Описанное выше поведение в зависимости от установленного значения свойства hibernate.event.merge.entity\_copy\_observer и ситуации может привести к потере данных! (Смотри дополнительную информацию в официальной доке).**

**Persist/Merge difference:**

* In case of **persist** method, if the entity that is to be managed in the persistence context, already exists in persistence context, the new one is ignored. (NOTHING happened)
* But in case of **merge** method, the entity that is already managed in persistence context will be replaced by the new entity (updated) and a copy of this updated entity will return back. (from now on any changes should be made on this returned entity if you want to reflect your changes in persistence context)

**Addition read:**

**-** [**https://temofeev.ru/info/articles/kak-rabotayut-metody-persist-merge-iz-jpa-i-metody-save-update-saveorupdate-iz-hibernate/**](https://temofeev.ru/info/articles/kak-rabotayut-metody-persist-merge-iz-jpa-i-metody-save-update-saveorupdate-iz-hibernate/)

**-** [**https://www.baeldung.com/hibernate-save-persist-update-merge-saveorupdate**](https://www.baeldung.com/hibernate-save-persist-update-merge-saveorupdate)

# CHECKING PERSISTENT STATE

Checking persistent state:

JPA, Hibernate:

**contains(entity);**

Checking laziness:

JPA:

PersistenceUnitUtil persistenceUnitUtil = entityManager.getEntityManagerFactory().getPersistenceUnitUtil();

boolean personInitialized =

persistenceUnitUtil.**isLoaded**(person);

boolean personBooksInitialized = persistenceUnitUtil.**isLoaded**(person.getBooks());

boolean personNameInitialized =

persistenceUnitUtil.**isLoaded**(person, "name");

или аналогичные методы класса PersistenceUtil persistenceUnitUtil = Persistence.getPersistenceUtil();

Hibernate:

boolean personInitialized =

Hibernate.**isInitialized**(person);

boolean personBooksInitialized =

Hibernate.**isInitialized**(person.getBooks());

boolean personNameInitialized =

Hibernate.**isPropertyInitialized**(person, "name");

# EVICTING ENTITIES

When the flush() method is called, the state of the entity is synchronized with the database. If you do not want this synchronization to occur, or if you are processing a huge number of objects and need to manage memory efficiently,

**evict()** – hibernate;

**detach()** – JPA;

methods can be used to remove the object and its collections from the first-level cache.

# CASCADING ENTITY STATE TRANSITIONS

JPA allows you to propagate the state transition from a parent entity to a child. For this purpose, the JPA javax.persistence.CascadeType defines various cascade types:

* **ALL**
* **PERSIST** - allows us to persist a child entity along with the parent one.
* **MERGE** - allows us to merge a child entity along with the parent one.
* **REMOVE** - allows us to remove a child entity along with the parent one.
* **REFRESH** - is used to propagate the refresh operation from a parent entity to a child. The refresh operation will discard the current entity state, and it will override it using the one loaded from the database.
* **DETACH** - is used to propagate the detach operation from a parent entity to a child.

Additionally, the CascadeType.ALL will propagate any Hibernate-specific operation, which is defined by the org.hibernate.annotations.CascadeType enum:

* **SAVE\_UPDATE** -
* **REPLICATE** - is to replicate both the parent and the child entities. The replicate operation allows you to synchronize entities coming from different sources of data.
* **LOCK** - allows us to reattach a parent entity along with its children to the currently running Persistence Context.

# @OnDelete CASCADE

While the previous cascade types propagate entity state transitions, the **@OnDelete** cascade is a DDL-level FK feature which allows you to remove a child record whenever the parent row is deleted.

# FLUSHING

**Flushing** is the process of synchronizing the state of the persistence context with the underlying database.

The flushing strategy is given by the [flushMode](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/Session.html#getFlushMode--):

* **AUTO -** This is the default mode, and it flushes the Session only if necessary.
* **COMMIT -** The Session tries to delay the flush until the current Transaction is committed, although it might flush prematurely too.

Hibernate supports additional strategies:

* **ALWAYS** - Flushes the Session before every query.
* **MANUAL** - The Session flushing is delegated to the application, which must call Session.flush() explicitly in order to apply the persistence context changes.

# DATABASE ACCESS

How to connect to your database so that it may connect on behalf of your application? This is ultimately the function of the **org.hibernate.engine.jdbc.connections.spi.ConnectionProvider** interface. Hibernate provides some out of the box implementations of this interface.

Applications should not have to configure a ConnectionProvider explicitly if using one of the Hibernate-provided implementations. Hibernate will internally determine which ConnectionProvider to use based on the following algorithm:

1. If hibernate.connection.provider\_class is set, it takes precedence
2. else if hibernate.connection.datasource is set → [Using DataSources](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#database-connectionprovider-datasource)
3. else if any setting prefixed by hibernate.c3p0. is set → [Using c3p0](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#database-connectionprovider-c3p0)
4. else if any setting prefixed by hibernate.proxool. is set → [Using Proxool](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#database-connectionprovider-proxool)
5. else if any setting prefixed by hibernate.hikari. is set → [Using HikariCP](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#database-connectionprovider-hikari)
6. else if any setting prefixed by hibernate.vibur. is set → [Using Vibur DBCP](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#database-connectionprovider-vibur)
7. else if any setting prefixed by hibernate.agroal. is set → [Using Agroal](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#database-connectionprovider-agroal)
8. else if hibernate.connection.url is set → [Using Hibernate’s built-in (and unsupported) pooling](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#database-connectionprovider-drivermanager)
9. else → [User-provided Connections](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#database-connectionprovider-provided)

# ConnectionProvider support for transaction isolation setting

All of the provided ConnectionProvider implementations, other than DataSourceConnectionProvider, support consistent setting of transaction isolation for all Connections obtained from the underlying pool. The value for **hibernate.connection.isolation** can be specified in one of 3 formats:

1. The integer value accepted at the JDBC level.
2. The name of the java.sql.Connection constant field representing the isolation you would like to use. For example, TRANSACTION\_REPEATABLE\_READ for [java.sql.Connection#TRANSACTION\_REPEATABLE\_READ](https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html#TRANSACTION_REPEATABLE_READ). Not that this is only supported for JDBC standard isolation levels, not for isolation levels specific to a particular JDBC driver.
3. A short-name version of the java.sql.Connection constant field without the TRANSACTION\_ prefix. For example, REPEATABLE\_READ for [java.sql.Connection#TRANSACTION\_REPEATABLE\_READ](https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html#TRANSACTION_REPEATABLE_READ). Again, this is only supported for JDBC standard isolation levels, not for isolation levels specific to a particular JDBC driver.

The SQL standard defines four Isolation levels:

* **READ\_UNCOMMITTED** (no lock on a table);
* **READ\_COMMITTED** (lock on commited data);
* **REPEATABLE\_READ** (lock on block of sql (which is selected by using select query));
* **SERIALIZABLE** (lock on a full table(on which Select query is fired)).

|  |  |  |  |
| --- | --- | --- | --- |
| **IsolationLevel** | **Dirty read** | **Non-repeatable read** | **Phantom read** |
| READ\_UNCOMMITTED | allowed | allowed | allowed |
| READ\_COMMITTED | prevented | allowed | allowed |
| REPEATABLE\_READ | prevented | prevented | allowed |
| SERIALIZABLE | prevented | prevented | Prevented |



Dirty read



Not-repeatable read



Phantom read

**Transaction isolation (additional read):**

1. <https://vladmihalcea.com/a-beginners-guide-to-transaction-isolation-levels-in-enterprise-java/>
2. <https://vladmihalcea.com/a-beginners-guide-to-acid-and-database-transactions/>
3. <https://stackoverflow.com/questions/16162357/transaction-isolation-levels-relation-with-locks-on-table>
4. An explanation of isolation levels locks - <https://retool.com/blog/isolation-levels-and-locking-in-relational-databases/>

# Connection handling

The connection handling mode is defined by the [PhysicalConnectionHandlingMode](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/resource/jdbc/spi/PhysicalConnectionHandlingMode.html) enumeration which provides the following strategies:

1. **IMMEDIATE\_ACQUISITION\_AND\_HOLD** - the Connection will be acquired as soon as the Session is opened and held until the Session is closed.
2. **DELAYED\_ACQUISITION\_AND\_HOLD** - the Connection will be acquired as soon as it is needed and then held until the Session is closed.
3. **DELAYED\_ACQUISITION\_AND\_RELEASE\_AFTER\_STATEMENT** - the Connection will be acquired as soon as it is needed and will be released after each statement is executed.
4. **DELAYED\_ACQUISITION\_AND\_RELEASE\_AFTER\_TRANSACTION** - the Connection will be acquired as soon as it is needed and will be released after each transaction is completed.

**hibernate.connection.handling\_mode** - the hibernate configure property forsettingconnection handling mode.

# Transaction type and connection handling

By default, the connection handling mode is given by the underlying transaction coordinator. There are two types of transactions:

1. **RESOURCE\_LOCAL (Transactions are managed by an application)**.
2. **JTA (Transactions are managed by an application server)**.

JPA implementations have the choice of managing transactions themselves (RESOURCE\_LOCAL), or having them managed by the application server's JTA implementation.

In most cases, RESOURCE\_LOCAL is fine. This would use basic JDBC-level transactions. The downside is that the transaction is local to the JPA persistence unit, so if you want a transaction that spans multiple persistence units (or other databases), then RESOURCE\_LOCAL may not be good enough.

JTA is also used for managing transactions across systems like JMS and JCA, but that's fairly exotic usage for most of us.To use JTA, you need support for it in your application server, and also support from the JDBC driver.

Read to understand!!! - (<https://stackoverflow.com/questions/1962525/persistence-unit-as-resource-local-or-jta>)

For RESOURCE\_LOCAL transactions, the connection handling mode is DELAYED\_ACQUISITION\_AND\_RELEASE\_AFTER\_TRANSACTION meaning that the database connection is acquired when needed and released after the current running transaction is either committed or rolled back.

However, because Hibernate needs to make sure that the default autocommit mode is disabled on the JDBC Connection when starting a new transaction, the Connection is acquired and the autocommit mode is set to false.

# LOCKING

In a relational database, locking refers to actions taken to prevent data from changing between the time it is read and the time is used.locking strategy can be either:

**Optimistic** – optimistic locking assumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect. Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction rolls back.

**Pessimistic** - Pessimistic locking assumes that concurrent transactions will conflict with each other, and requires resources to be locked after they are read and only unlocked after the application has finished using the data.

Concurrency modes (optimistic and pesimistic locks) is additional level of security over isolation levels (?).

* <https://stackoverflow.com/questions/22646226/how-are-locking-mechanisms-pessimistic-optimistic-related-to-database-transact>
* <https://apacheignite.readme.io/docs/concurrency-modes-and-isolation-levels>

# Optimistic locking (OL)

OL guarantees some isolation, but scales well and works particularly well in *read-often-write-sometimes* situations. Hibernate provides two different mechanisms for storing versioning information:

* **Dedicated version number;**
* **Timestamp.**

A version or timestamp property can never be null for a detached instance. *Hibernate detects any instance with a null version or timestamp as transient, regardless of other unsaved-value strategies that you specify. Declaring a nullable version or timestamp property is an easy way to avoid problems with transitive reattachment in Hibernate, especially useful if you use assigned identifiers or composite keys.* (Unsaved-value strategy - <https://stackoverflow.com/questions/8769308/what-is-the-need-for-an-unsaved-value-attribute-in-hibernate>)

To enable optimistic locking simply add the **javax.persistence.Version** to the persistent attribute that defines the optimistic locking value.

**JPA**:

* **int** or **Integer**;
* **short** or **Short**;
* **long** or **Long**;
* **java.sql.Timestamp**.

**Hibernate**:

* Java 8 Date/Time types, such as **Instant.**

Your application is forbidden from altering the version number set by Hibernate. To artificially increase the version number, see the documentation for properties:

* **LockModeType.OPTIMISTIC\_FORCE\_INCREMENT** or
* **LockModeType.PESSIMISTIC\_FORCE\_INCREMENT**

check in the Hibernate Entity Manager reference documentation.

If the version number is generated by the database, such as a trigger, use the annotation **@org.hibernate.annotations.Generated(GenerationTime.ALWAYS)** on the version attribute.

По умолчанию измение любого свойства entityприводит к изменению версии. Что изменение свойства не вызывало изменение версии используется аннотация **@OptimisticLock(excluded = true)**.

**Timestamp**

Timestamps are a less reliable way of optimistic locking than version numbers but can be used by applications for other purposes as well. Timestamping is automatically used if you the @Version annotation on a Date or Calendar property type.

Hibernate can retrieve the timestamp value from the database or the JVM:

* **Database** - **@Source(value = SourceType.DB)** (Default).

Or **@Generated(GenerationTime.ALWAYS)**

* **JVM** - **@Source(value = SourceType.VM)**

Hibernate supports a form of optimistic locking that does not require a dedicated "version attribute".This is achieved through the use of the [**@OptimisticLocking**](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/OptimisticLocking.html) annotation which defines a single attribute of type [**org.hibernate.annotations.OptimisticLockType**](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/OptimisticLockType.html).

There are 4 available OptimisticLockTypes:

* **NONE -** optimistic locking is disabled even if there is a @Version annotation present;
* **VERSION(the default) -** performs optimistic locking based on a @Version as described above;
* **ALL -** performs optimistic locking based on *all* fields as part of an expanded WHERE clause restriction for the UPDATE/DELETE SQL statements;
* **DIRTY -** performs optimistic locking based on *dirty* fields as part of an expanded WHERE clause restriction for the UPDATE/DELETE SQL statements.

# Pessimistic locking (PL)

Long before JPA 1.0, Hibernate already defined various explicit locking strategies through its LockMode enumeration. JPA comes with its own [LockModeType](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/LockModeType.html) enumeration which defines similar strategies as the Hibernate-native LockMode.

| **LockModeType** | **LockMode** | **Description** |
| --- | --- | --- |
| NONE | NONE | The absence of a lock. All objects switch to this lock mode at the end of a Transaction. Objects associated with the session via a call to update() or saveOrUpdate() also start out in this lock mode. |
| READ and OPTIMISTIC | READ | The entity version is checked towards the end of the currently running transaction. |
| WRITE and OPTIMISTIC\_FORCE\_INCREMENT | WRITE | The entity version is incremented automatically even if the entity has not changed. |
| PESSIMISTIC\_FORCE\_INCREMENT | PESSIMISTIC\_FORCE\_INCREMENT | The entity is locked pessimistically and its version is incremented automatically even if the entity has not changed. |
| PESSIMISTIC\_READ | PESSIMISTIC\_READ | The entity is locked pessimistically using a shared lock if the database supports such a feature. Otherwise, an explicit lock is used. |
| PESSIMISTIC\_WRITE | PESSIMISTIC\_WRITE, UPGRADE | The entity is locked using an explicit lock. |
| PESSIMISTIC\_WRITE with a javax.persistence.lock.timeout setting of 0 | UPGRADE\_NOWAIT | The lock acquisition request fails fast if the row s already locked. |
| PESSIMISTIC\_WRITE with a javax.persistence.lock.timeout setting of -2 | UPGRADE\_SKIPLOCKED | The lock acquisition request skips the already locked rows. It uses a SELECT …​ FOR UPDATE SKIP LOCKED in Oracle and PostgreSQL 9.5, or SELECT …​ with (rowlock, updlock, readpast) in SQL Server. |

The explicit user request mentioned above occurs as a consequence of any of the following actions:

* a call to **Session.load()**, specifying a **LockMode**.
* a call to **Session.lock()**.
* a call to **Query.setLockMode()**.

If you call **Session.load()** with option

* **UPGRADE;**
* **UPGRADE\_NOWAIT;**
* **UPGRADE\_SKIPLOCKED.**

and the requested object is not already loaded by the session, the object is loaded using **SELECT …​ FOR UPDATE**.

If you call **load()** for an object that is already loaded with a less restrictive lock than the one you request, Hibernate calls **lock()** for that object.

**Session.lock()** performs a version number check if the specified lock mode is

* **READ;**
* **UPGRADE;**
* **UPGRADE\_NOWAIT;**
* **UPGRADE\_SKIPLOCKED.**

In the case of

* **UPGRADE;**
* **UPGRADE\_NOWAIT;**
* **UPGRADE\_SKIPLOCKED.**

the **SELECT …​ FOR UPDATE** syntax is used.

If the requested lock mode is not supported by the database, Hibernate uses an appropriate alternate mode instead of throwing an exception. This ensures that applications are portable.

# JPA locking query hints

JPA 2.0 introduced two query hints:

1. **javax.persistence.lock.timeout** - it gives the number of milliseconds a lock acquisition request will wait before throwing an exception
2. **javax.persistence.lock.scope** - defines the [*scope*](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/PessimisticLockScope.html) of the lock acquisition request. The scope can either be NORMAL (default value) or EXTENDED. The EXTENDED scope will cause a lock acquisition request to be passed to other owned table structured (e.g. @Inheritance(strategy=InheritanceType.JOINED), @ElementCollection)

Not all JDBC database drivers support setting a timeout value for a locking request. If not supported, the Hibernate dialect ignores this query hint. The javax.persistence.lock.scopeis [not yet supported](https://hibernate.atlassian.net/browse/HHH-9636) as specified by the JPA standard.

# FETCHING

There are a number of scopes for defining fetching:

1. **STATIC** *-* Static definition of fetching strategies is done in the mappings..
2. **SELECT** – Performs a separate SQL select to load the data. This can either be EAGER (the second select is issued immediately) or LAZY (the second select is delayed until the data is needed). This is the strategy generally termed N+1.
3. **JOIN** – Inherently an EAGER style of fetching. The data to be fetched is obtained through the use of an SQL outer join.
4. **BATCH** – Performs a separate SQL select to load a number of related data items using an IN-restriction as part of the SQL WHERE-clause based on a batch size. Again, this can either be EAGER (the second select is issued immediately) or LAZY (the second select is delayed until the data is needed).
5. **SUBSELECT** – Performs a separate SQL select to load associated data based on the SQL restriction used to load the owner. Again, this can either be EAGER (the second select is issued immediately) or LAZY (the second select is delayed until the data is needed).
6. **DYNAMIC** (sometimes referred to as runtime) – The dynamic definition is really use-case centric. There are multiple ways to define dynamic fetching:
7. **FETCH PROFILES** – defined in mappings, but can be enabled/disabled on the Session.
8. **HQL / JPQL** – both Hibernate and JPA Criteria queries have the ability to specify fetching, specific to said query.
9. **ENTITY GRAPHS** – starting in Hibernate 4.2 (JPA 2.1), this is also an option.

**The statically-defined fetch strategies are used in the absence of any dynamically defined strategies!**

# FETCH TYPES

The **FetchType** method defines two strategies for fetching data from the database:

* **FetchType.EAGER:** The persistence provider must load the related annotated field or property. This is the default behavior for *@Basic, @ManyToOne*, and *@OneToOne*annotated fields.
* **FetchType.LAZY:** The persistence provider should load data when it's first accessed, but can be loaded eagerly. This is the default behavior for *@OneToMany, @ManyToMany* and *@ElementCollection-*annotated fields.

If you forget to JOIN FETCH all EAGER associations, Hibernate is going to issue a secondary select for each and every one of those which, in turn, can lead to N+1 query issues.For this reason, you should prefer LAZY associations.

The Hibernate recommendation is to statically mark all associations lazy and to use dynamic fetching strategies for eagerness.

# Dynamic fetching via JPA entity graph

JPA 2.1 introduced entity graph so the application developer has more control over fetch plans. It has two modes to choose from:

1. **Fetch graph** – In this case, all attributes specified in the entity graph will be treated as FetchType.EAGER, and all attributes not specified will ALWAYS be treated as FetchType.LAZY.
2. **Load graph** – In this case, all attributes specified in the entity graph will be treated as FetchType.EAGER, but attributes not specified use their static mapping specification.

JPA Entity Graph (Baeldung) – <https://www.baeldung.com/jpa-entity-graph>.

A sub-graph is used to control the fetching of sub-attributes of the AttributeNode it is applied to. It is generally defined via the [@NamedSubgraph](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/NamedSubgraph.html) annotation.

Multiple entity graphs can be combined into a single "super graph" that acts as a union.

# Dynamic fetching via Hibernate profiles

@Entity(name = "Employee")

@FetchProfile(

name = "employee.projects",

fetchOverrides = {

@FetchProfile.FetchOverride(

entity = Employee.class,

association = "projects",

mode = FetchMode.JOIN

)

}

)

session.enableFetchProfile( "employee.projects" );

Employee employee = session.bySimpleNaturalId( Employee.class ).load( username );

# Static fetching via Hibernate specific @Fetch annotation

Besides the FetchType.LAZY or FetchType.EAGER JPA annotations, you can also use the Hibernate-specific **@Fetch** annotation that accepts one of the following **FetchModes**:

**SELECT** – The association is going to be fetched lazily using a secondary select for each individual entity, collection, or join load. It’s equivalent to JPA FetchType.LAZY fetching strategy. Can lead to N+1 query issues.

**SUBSELECT** – Available for collections only. When accessing a non-initialized collection, this fetch mode will trigger loading all elements of all collections of the same role for all owners associated with the persistence context using a single secondary select.

**JOIN** – Use an outer join to load the related entities, collections or joins when using direct fetching. It’s equivalent to JPA FetchType.EAGER fetching strategy.

When using JPQL query the FetchMode.JOIN strategy would be overridden by the query fetching directive. To fetch multiple relationships with a JPQL query, the JOIN FETCH directive must be used instead. Therefore, FetchMode.JOIN is useful for when entities are fetched directly, via their identifier or natural-id. Also, the FetchMode.JOIN acts as a FetchType.EAGER strategy. Even if we mark the association as FetchType.LAZY, the FetchMode.JOIN will load the association eagerly.

Dynamic fetching overrides static one!

# Static fetching via Hibernate specific @LazyCollection anotaion

The [**@LazyCollection**](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/LazyCollection.html) annotation is used to specify the lazy fetching behavior of a given collection. The possible values are given by the [**LazyCollectionOption**](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/LazyCollectionOption.html) enumeration:

**TRUE** – Load it when the state is requested.

***DEPRECATED!*** *Should be using the JPA* [*FetchType*](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/FetchType.html) *attribute of the* [*@ElementCollection*](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#annotations-jpa-elementcollection)*,* [*@OneToMany*](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#annotations-jpa-onetomany)*, or* [*@ManyToMany*](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#annotations-jpa-manytomany) *collection.*

**FALSE**– Eagerly load it.

***DEPRECATED!*** *Should be using the JPA* [*FetchType*](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/FetchType.html) *attribute of the* [*@ElementCollection*](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#annotations-jpa-elementcollection)*,* [*@OneToMany*](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#annotations-jpa-onetomany)*, or* [*@ManyToMany*](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#annotations-jpa-manytomany) *collection.*

**EXTRA** – Prefer extra queries over full collection loading.The EXTRA value has no equivalent in the JPA specification. Each element is fetched individually using a secondary query.

*Using EXTRA Lazy Collections with Hibernate is a bad idea since it can lead to N+1 query issues and cause performance problems!*

*[https://stackoverflow.com/questions/12928402/what-is-the-use-of-the-hibernate-lazycollection-annotation/40132098#40132098](https://stackoverflow.com/questions/12928402/what-is-the-use-of-the-hibernate-lazycollection-annotation/40132098" \l "40132098)*

# BATCHING

What is batching in hibernate (simple) - <https://stackoverflow.com/questions/45670583/how-hibernate-batch-insert-works>

Hibernate can leverage JDBC batching. The following settings control this behavior:

1. **hibernate.jdbc.batch\_size** – controls the maximum number of statements Hibernate will batch together before asking the driver to execute the batch. Zero or a negative number disables this feature.
2. **…**

Since version 5.2, Hibernate allows overriding the global JDBC batch size given by the **hibernate.jdbc.batch\_size** configuration property on a per Session basis:

entityManager.unwrap(Session.class).setJdbcBatchSize(10);

Hibernate disables insert batching at the JDBC level transparently if you use an identity identifier generator.

# Session scroll

When you retrieve and update data, flush() and clear() the session regularly. In addition, use method scroll() to take advantage of server-side cursors for queries that return many rows of data.

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

scrollableResults = session

.createQuery("select p from Person p")

.setCacheMode(CacheMode.IGNORE)

.scroll(ScrollMode.FORWARD\_ONLY);

int count = 0;

while (scrollableResults.next()) {

Person Person = (Person) scrollableResults.get(0);

processPerson(Person);

if ( ++count % batchSize == 0) {

//flush a batch of updates and release memory:

entityManager.flush();

entityManager.clear();

}

}

If left unclosed by the application, Hibernate will automatically close the underlying resources (e.g. ResultSet and PreparedStatement) used internally by the ScrollableResults when the current transaction is ended (either commit or rollback). However, it is good practice to close the ScrollableResults explicitly.

# StatelessSession

StatelessSession is a command-oriented API provided by Hibernate. Use it to stream data to and from the database in the form of detached objects. A StatelessSession has no persistence context associated with it and does not provide many of the higher-level lifecycle semantics.

Some of the things not provided by a StatelessSession include:

* a first-level cache
* interaction with any second-level or query cache
* transactional write-behind or automatic dirty checking

Limitations of StatelessSession:

* Operations performed using a stateless session never cascade to associated instances.
* Collections are ignored by a stateless session.
* Lazy loading of associations is not supported.
* Operations performed via a stateless session bypass Hibernate’s event model and interceptors.
* Due to the lack of a first-level cache, Stateless sessions are vulnerable to data aliasing effects.
* A stateless session is a lower-level abstraction that is much closer to the underlying JDBC.

StatelessSession statelessSession = null;

Transaction txn = null;

ScrollableResults scrollableResults = null;

try {

SessionFactory sessionFactory = entityManagerFactory().unwrap( SessionFactory.class );

statelessSession = sessionFactory.openStatelessSession();

txn = statelessSession.getTransaction();

txn.begin();

scrollableResults = statelessSession

.createQuery( "select p from Person p" )

.scroll(ScrollMode.FORWARD\_ONLY);

while ( scrollableResults.next() ) {

Person Person = (Person) scrollableResults.get( 0 );

processPerson(Person);

statelessSession.update( Person );

}

txn.commit();

} catch (RuntimeException e) {

if ( txn != null && txn.getStatus() == TransactionStatus.ACTIVE) txn.rollback();

throw e;

} finally {

if (scrollableResults != null) {

scrollableResults.close();

}

if (statelessSession != null) {

statelessSession.close();

}

}

The Customer instances returned by the query are immediately detached. They are never associated with any persistence context.

The insert(), update(), and delete() operations defined by the StatelessSession interface operate directly on database rows. They cause the corresponding SQL operations to be executed immediately. They have different semantics from the save(), saveOrUpdate(), and delete() operations defined by the Session interface.

<https://stackoverflow.com/questions/4384515/need-enough-information-about-stateless-session-in-hibernate>

# CACHING

Hibernate can integrate with various caching providers for the purpose of caching data outside the context of a particular Session.

**Second level cache**

Be aware that Hibernate caches are not aware of changes made to the persistent store by other applications. To address this limitation, you can configure a TTL (Time To Live) retention policy at the second-level cache region level so that the underlying cache entries expire regularly.

**org.hibernate.cache.spi.RegionFactory**defines the integration between Hibernate and a pluggable caching provider. **hibernate.cache.region.factory\_class** is used to declare the provider to use.

Hibernate comes with built-in support for the Java caching standard:

* [JCache](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#caching-provider-jcache).

and also two popular caching libraries:

* [Ehcache](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#caching-provider-ehcache);
* [Infinispan](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#caching-provider-infinispan).

# Configuring second-level caching

* **hibernate.cache.use\_second\_level\_cache** – enable or disable second level caching overall. By default, if the currently configured [RegionFactory](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/cache/spi/RegionFactory.html) is not the **NoCachingRegionFactory**, then the second-level cache is going to be enabled. Otherwise, the second-level cache is disabled.
* **hibernate.cache.use\_query\_cache** – enable or disable second level caching of query results. The default is false.
* **hibernate.cache.query\_cache\_factory** – query result caching is handled by a special contract that deals with staleness-based invalidation of the results. The default implementation does not allow stale results at all. Use this for applications that would like to relax that. Names an implementation of org.hibernate.cache.spi.TimestampsCacheFactory.
* **hibernate.cache.use\_minimal\_puts** – optimizes second-level cache operations to minimize writes, at the cost of more frequent reads. Providers typically set this appropriately.
* **hibernate.cache.region\_prefix** – defines a name to be used as a prefix to all second-level cache region names.
* **hibernate.cache.default\_cache\_concurrency\_strategy** – in Hibernate second-level caching, all regions can be configured differently including the concurrency strategy to use when accessing that particular region. This setting allows defining a default strategy to be used. This setting is very rarely required as the pluggable providers do specify the default strategy to use. Valid values include:
* read-only,
* read-write,
* nonstrict-read-write,
* transactional
* **hibernate.cache.use\_structured\_entries** – if true, forces Hibernate to store data in the second-level cache in a more human-friendly format. Can be useful if you’d like to be able to "browse" the data directly in your cache, but does have a performance impact.
* **hibernate.cache.auto\_evict\_collection\_cache** – enables or disables the automatic eviction of a bidirectional association’s collection cache entry when the association is changed just from the owning side. This is disabled by default, as it has a performance impact to track this state. However, if your application does not manage both sides of bidirectional association where the collection side is cached, the alternative is to have stale data in that collection cache.
* **hibernate.cache.use\_reference\_entries** – enable direct storage of entity references into the second level cache for read-only or immutable entities.
* **hibernate.cache.keys\_factory** – when storing entries into the second-level cache as a key-value pair, the identifiers can be wrapped into tuples <entity type, tenant, identifier> to guarantee uniqueness in case that second-level cache stores all entities in single space. These tuples are then used as keys in the cache. When the second-level cache implementation (incl. its configuration) guarantees that different entity types are stored separately and multi-tenancy is not used, you can omit this wrapping to achieve better performance. Currently, this property is only supported when Infinispan is configured as the second-level cache implementation. Valid values are:
* default (wraps identitifers in the tuple)
* simple (uses identifiers as keys without any wrapping)
* fully qualified class name that implements org.hibernate.cache.spi.CacheKeysFactory

# Configuring second-level cache mappings

By default, entities are not part of the second level cache. However, you can override this by setting the **shared-cache-mode** element in your ***persistence.xml*** file or by using the ***javax.persistence.sharedCache.mode*** property in your configuration file:

1. **Enable\_selective** (Default, recommended) –entities are not cached unless explicitly marked as cacheable (with the [@Cacheable](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Cacheable.html) annotation).
2. **Disable\_selective**– entities are cached unless explicitly marked as non-cacheable.
3. **All**–entities are always cached even if marked as non-cacheable.
4. **None**–no entity is cached even if marked as cacheable. This option can make sense to disable second-level cache altogether.

The **cache concurrency strategy** used by default can be set globally via the **hibernate.cache.default\_cache\_concurrency\_strategy** configuration property:

1. **Read-only** – if your application needs to read, but not modify, instances of a persistent class, a read-only cache is the best choice. Application can still delete entities and these changes should be reflected in second-level cache so that the cache does not provide stale entities. Implementations may use performance optimizations based on the immutability of entities.
2. **Read-write** – if the application needs to update data, a read-write cache might be appropriate. This strategy provides consistent access to single entity, but not a serializable transaction isolation level; e.g. when TX1 reads looks up an entity and does not find it, TX2 inserts the entity into cache and TX1 looks it up again, the new entity can be read in TX1.
3. **Nonstrict-read-write** – similar to read-write strategy but there might be occasional stale reads upon concurrent access to an entity. The choice of this strategy might be appropriate if the application rarely updates the same data simultaneously and strict transaction isolation is not required. Implementations may use performance optimizations that make use of the relaxed consistency guarantee.
4. **Transactional** – provides serializable transaction isolation level.

Rather than using a global setting, it is recommended to define the cache concurrency strategy on a per entity basis. Use the [**@org.hibernate.annotations.Cache**](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/Cache.html) annotation for this purpose. The **@Cache** annotation define three attributes:

1. **Usage** – defines the CacheConcurrencyStrategy.
2. **Region** – defines a cache region where entries will be stored.
3. **Include** – if lazy properties should be included in the second level cache. The default value is ***all*** so lazy properties are cacheable. The other possible value is ***non-lazy*** so lazy properties are not cacheable.

# Entity inheritance and second-level cache mapping

Traditionally, when using entity inheritance, Hibernate required an entity hierarchy to be either cached entirely or not cached at all. Therefore, if you wanted to cache a subclass belonging to a given entity hierarchy, the JPA @Cacheable and the Hibernate-specific @Cache annotations **would have to be declared at the root-entity level only**. (Хотя в JPA по другому, и в подклассе можно переопределить поведение, и с версии 5.3 hibernate поддерживает это, но рекамендуется анотировать только root-entity).

# Entity cache

Hibernate stores cached entities in a dehydrated form, which is similar to the database representation. Aside from the foreign key column values of the @ManyToOne or @OneToOne child-side associations, entity relationships are not stored in the cache.

The Hibernate second-level cache can also load entities by their [natural id](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#naturalid).

# Collection cache

Hibernate can also cache collections, and the @Cache annotation must be on added to the collection property.

If the collection is made of value types (basic or embeddables mapped with @ElementCollection), the collection is stored as such. If the collection contains other entities (@OneToMany or @ManyToMany), the collection cache entry will store the entity identifiers only.

The collection cache is not write-through so any modification will trigger a collection cache entry invalidation. On a subsequent access, the collection will be loaded from the database and re-cached.

# Query cache

Caching of query results introduces some overhead in terms of your applications normal transactional processing. For example, if you cache results of a query against Person, Hibernate will need to keep track of when those results should be invalidated because changes have been committed against any Person entity.

That, coupled with the fact that most applications simply gain no benefit from caching query results, leads Hibernate to disable caching of query results by default.

For entity queries, the query cache does not cache the state of the actual entities. Instead, it stores the entity identifiers, and when the query result is fetched from the cache, the entity state is going to be loaded from the second-level cache entity regions.

Just as with collection caching, the query cache should always be used in conjunction with the second-level cache for those entities expected to be cached as part of a query result cache.

For projection queries, the query cache stores the dehydrated entity state (e.g. Object[]) associated with the underlying JDBC ResultSet.

# Managing the cached data

Traditionally, Hibernate defined the [CacheMode](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/CacheMode.html) enumeration to describe the ways of interactions with the cached data. JPA split cache modes by storage ([CacheStoreMode](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/CacheStoreMode.html)) and retrieval ([CacheRetrieveMode](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/CacheRetrieveMode.html)).

The relationship between Hibernate and JPA cache modes can be seen in the following table:

| Table 5. Cache modes relationships | | |
| --- | --- | --- |
| **Hibernate** | **JPA** | **Description** |
| CacheMode.NORMAL | CacheStoreMode.USE and CacheRetrieveMode.USE | Default. Reads/writes data from/into the cache |
| CacheMode.REFRESH | CacheStoreMode.REFRESH and CacheRetrieveMode.BYPASS | Doesn’t read from cache, but writes to the cache upon loading from the database |
| CacheMode.PUT | CacheStoreMode.USE and CacheRetrieveMode.BYPASS | Doesn’t read from cache, but writes to the cache as it reads from the database |
| CacheMode.GET | CacheStoreMode.BYPASS and CacheRetrieveMode.USE | Read from the cache, but doesn’t write to cache |
| CacheMode.IGNORE | CacheStoreMode.BYPASS and CacheRetrieveMode.BYPASS | Doesn’t read/write data from/into the cache |

**Evicting cache entries:**

Because the second level cache is bound to the either

* **EntityManagerFactory**
* or **SessionFactory**

cache eviction must be done through these two interfaces.

# Caching statistics

If you enable the hibernate.generate\_statistics configuration property, Hibernate will expose a number of metrics via SessionFactory.getStatistics(). Hibernate can even be configured to expose these statistics via JMX.

This way, you can get access to the [Statistics](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/stat/Statistics.html) class which comprises all sort of second-level cache metrics.

# Doc references

Hibernate Second-Level Cache (Baeldung) – <https://www.baeldung.com/hibernate-second-level-cache>.

# INTERCEPTORS AND EVENTS

# Interceptors

The **org.hibernate.Interceptor** interface provides callbacks **from the session to the application**, allowing the application to inspect and/or manipulate properties of a persistent object before it is saved, updated, deleted or loaded.

An Interceptor can be:

* Session-scoped;
* SessionFactory-scoped.

# Native event system

If you have to react to particular events in the persistence layer, you can also use the Hibernate *event* architecture. The event system can be used in place of or in addition to interceptors.

Many methods of the Session interface correlate to an event type. The full range of defined event types is declared as enum values on org.hibernate.event.spi.EventType. When a request is made of one of these methods, the Session generates an appropriate event and passes it to the configured event listener(s) for that type.

# JPA callbacks

JPA also defines a more limited set of callbacks through annotations.

| **Type** | **Description** |
| --- | --- |
| @PrePersist | Executed before the entity manager persist operation is actually executed or cascaded. This call is synchronous with the persist operation. |
| @PreRemove | Executed before the entity manager remove operation is actually executed or cascaded. This call is synchronous with the remove operation. |
| @PostPersist | Executed after the entity manager persist operation is actually executed or cascaded. This call is invoked after the database INSERT is executed. |
| @PostRemove | Executed after the entity manager remove operation is actually executed or cascaded. This call is synchronous with the remove operation. |
| @PreUpdate | Executed before the database UPDATE operation. |
| @PostUpdate | Executed after the database UPDATE operation. |
| @PostLoad | Executed after an entity has been loaded into the current persistence context or an entity has been refreshed. |

There are two available approaches defined for specifying callback handling:

* The first approach is to annotate methods on the entity itself to receive notifications of a particular entity lifecycle event(s).
* The second is to use a separate entity listener class. An entity listener is a stateless class with a no-arg constructor. The callback annotations are placed on a method of this class instead of the entity class. The entity listener class is then associated with the entity using the javax.persistence.EntityListeners annotation

# Default entity listeners

* The JPA specification allows you to define a default entity listener which is going to be applied for every entity in that particular system. Default entity listeners can only be defined in XML mapping files.

# HQL, JPQL

**HQL** - Hibernate Query Language.

**JPQL** - Java Persistence Query Language.

JPQL is a heavily-inspired-by subset of HQL. A JPQL query is always a valid HQL query, the reverse is not true, however.

Both HQL and JPQL are non-type-safe ways to perform query operations.

Since 5.2, the Hibernate Session interface extends the JPA EntityManager interface. For this reason, the query API was also merged, and now the Hibernate org.hibernate.query.Query interface extends the JPA javax.persistence.Query.

# JPA query API

In JPA, the query is represented by

* **javax.persistence.Query** or
* **javax.persistence.TypedQuery**

as obtained from the **EntityManager.**

Many of the settings controlling the execution of the query are defined as hints. JPA defines some standard hints (like timeout in the example), but most are provider specific:

* **javax.persistence.query.timeout** - Defines the query timeout, in milliseconds.
* **javax.persistence.fetchgraph**- Defines a fetchgraph EntityGraph. Attributes explicitly specified as AttributeNodes are treated as FetchType.EAGER (via join fetch or subsequent select). For details, see the EntityGraph discussions in [Fetching](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#fetching).
* **javax.persistence.loadgraph** – Defines a loadgraph EntityGraph. Attributes explicitly specified as AttributeNodes are treated as FetchType.EAGER (via join fetch or subsequent select). Attributes that are not specified are treated as FetchType.LAZY or FetchType.EAGER depending on the attribute’s definition in metadata. For details, see the EntityGraph discussions in [Fetching](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#fetching).
* **org.hibernate.cacheMode** - Defines the CacheMode to use. See org.hibernate.query.Query#setCacheMode.
* **org.hibernate.cacheable** - Defines whether the query is cacheable. true/false. See org.hibernate.query.Query#setCacheable.
* **org.hibernate.cacheRegion** - For queries that are cacheable, defines a specific cache region to use. See org.hibernate.query.Query#setCacheRegion.
* **org.hibernate.comment** - Defines the comment to apply to the generated SQL. See org.hibernate.query.Query#setComment.
* **org.hibernate.fetchSize** - Defines the JDBC fetch-size to use. See org.hibernate.query.Query#setFetchSize.
* **org.hibernate.flushMode** - Defines the Hibernate-specific FlushMode to use. See org.hibernate.query.Query#setFlushMode. If possible, prefer using javax.persistence.Query#setFlushMode instead.
* **org.hibernate.readOnly** - Defines that entities and collections loaded by this query should be marked as read-only. See org.hibernate.query.Query#setReadOnly.

JPA defines a simplified set of parameter binding methods. It supports:

* Setting the parameter value by name;
* Setting the parameter value by position;
* Specialized form for Calendar/Date types additionally accepting a TemporalType.

In terms of execution, JPA Query offers 3 different methods for retrieving a result set:

* **Query.getResultList()** - executes the select query and returns back the list of results.
* **Query.getResultStream()** - executes the select query and returns back a Stream over the results.
* **Query.getSingleResult()** - executes the select query and returns a single result. If there were more than one result an exception is thrown.

# Hibernate query API

In terms of execution, Hibernate offers 4 different methods. The 2 most commonly used are

* **Query.list** - executes the select query and returns back the list of results.
* **Query.uniqueResult** - executes the select query and returns the single result. If there were more than one result an exception is thrown.

# Query scrolling

Hibernate offers additional, specialized methods for scrolling the query and handling results using a **server-side cursor**. **Query.scroll()** works in tandem with the JDBC notion of a scrollable ResultSet. The method can accept the ScrollMode parameter.

If you plan to use **Query.scroll()** with collection fetches it is important that your query explicitly order the results so that the JDBC results contain the related rows sequentially.

Hibernate also supports **Query.iterate()**, which is intended for loading entities when it is known that the loaded entries are already stored in the second-level cache.

Since 5.2, Hibernate offers support for returning a Stream which can be later used to transform the underlying ResultSet.

Close ResultSets and Streams!

# Query streaming

Since version 2.2, the JPA Query interface offers support for returning a Stream via the **getResultStream** method.

# Case sensitivity

Queries are case-insensitive, exception:

* Names of Java classes;
* Names of properties.

Use lowercase keywords as a convention!

# Statement types

Both HQL and JPQL allow the following statements to be performed:

* **SELECT**;
* **UPDATE**;
* **DELETE**.

HQL additionally allows **INSERT** statements, in a form similar to a SQL **INSERT FROM SELECT**.

Caution should be used when executing bulk update or delete operations because they may result in inconsistencies between the database and the entities in the active persistence context. In general, bulk update and delete operations should only be performed within a transaction in a new persistence context or before fetching or accessing entities whose state might be affected by such operations.

# Select

The BNF for SELECT statements in HQL is:

**select\_statement**::=

[select\_clause]

from\_clause

[where\_clause]

[groupby\_clause]

[having\_clause]

[orderby\_clause]

# Update

The BNF for UPDATE statements is the same in HQL and JPQL:

**update\_statement** ::= **update\_clause** [where\_clause]

**update\_clause** ::=

UPDATE entity\_name [[AS] identification\_variable]

SET **update\_item**{, **update\_item**}\*

**update\_item** ::=

[identification\_variable.]{state\_field | single\_valued\_object\_field} = **new\_value**

**new\_value** ::= scalar\_expression | simple\_entity\_expression | NULL

**UPDATE** statements, by default, do not affect the **version** or the **timestamp** attribute values for the affected entities. To force Hibernate to set the **version** or **timestamp** attribute values add the **VERSIONED** keyword after the **UPDATE** keyword.

Versioned updates is a Hibernate-specific feature. Custom version types, **org.hibernate.usertype.UserVersionType**, are not allowed in conjunction with an **update versioned** statement.

The **int** value returned by the **executeUpdate()** method indicates the number of entities affected by the operation. This may or may not correlate to the number of rows affected in the database (see JoinedTable hierarchy strategy for example).

Neither **UPDATE** nor **DELETE** statements allow any joins.

# Delete

The BNF for DELETE statements is the same in HQL and JPQL:

**delete\_statement** ::= **delete\_clause** [where\_clause]

**delete\_clause** ::= DELETE FROM entity\_name [[AS] identification\_variable]

# Insert

The BNF for an HQL INSERT statement is:

**insert\_statement** ::= **insert\_clause** select\_statement

**insert\_clause** ::= INSERT INTO entity\_name (**attribute\_list**)

**attribute\_list** ::= state\_field[, state\_field ]\*

There is **NO** JPQL equivalent to HQL-style INSERT statements.

select pr1 from by.pva.hibernate.part01.hql\_jpql.domain\_model.Person pr1, by.pva.hibernate.part01.hql\_jpql.domain\_model.Person pr2

# Distinct

For JPQL and HQL, **DISTINCT** has two meanings:

1. It can be passed to the database so that duplicates are removed from a result set.
2. It can be used to filter out the same parent entity references when join fetching a child collection.

**Important about distinct** - https://in.relation.to/2016/08/04/introducing-distinct-pass-through-query-hint/.

# CRITERIA API

The**org.hibernate.Criteria API**is deprecated. Use **javax.persistence.criteria.CriteriaQuery.**

Criteria queries are essentially an object graph. The first step in performing a criteria query is building this graph usingthe **javax.persistence.criteria.CriteriaBuilder** interface. Its role is that of a factory for all the individual pieces of the criteria.

You obtain a CriteriaBuilder instance by calling the **getCriteriaBuilder**() method of either:

* javax.persistence.**EntityManagerFactory**;
* javax.persistence.**EntityManager**.

To obtain a **javax.persistence.criteria.CriteriaQuery** use one of the three methods on CriteriaBuilder:

* <T> CriteriaQuery<T>**createQuery**(Class<T> resultClass);
* CriteriaQuery<Tuple>**createTupleQuery**();
* CriteriaQuery<Object>**createQuery**().

# Native queries

# 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/>.
7. RESOURCE\_LOCAL Vs. JTA - <https://dzone.com/articles/resource-local-vs-jta-transaction-types-and-payara>.
8. Deleting objects with Hibernate - <https://www.baeldung.com/delete-with-hibernate>.

<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/>

New links

<https://howtodoinjava.com/hibernate/hibernate-c3p0-connection-pool-configuration-tutorial/>

<https://eng.fitbit.com/instrumenting-hibernate-connection-providers/>

https://stackoverflow.com/questions/43198457/when-to-close-hibernate-session