JPA & Hibernate

Автор:
Эльдар Суйналиев

При участии:
Андрей Агалаков
Роман Евсеев
Максим Казаков
Антон Таврель

2020 год

Оглавление

Введение

В настоящем материале используется официальная документация и руководства Oracle, Java Persistence API, фреймворка Hibernate. Практические примеры преимущественно получены с сайтов, пользующихся популярностью и уважением в мировом сообществе разработчиков: Baeldung, Vlad Mihalcea, Thorben Janssen и другие.

Использованные версии документаций:
Java Persistence API версии 2.2 от 2017.07.17
Java Enterprise Edition (Java EE) 8 Tutorial
Hibernate ORM 5.4.17.Final User Guide
Hibernate JavaDoc (5.4.17.Final)

Если Вы заметите неточность или ошибку, либо захотите дополнить ответ - пиши мне, и мы всё обсудим.

1. Что такое ORM? Что такое JPA? Что такое Hibernate?

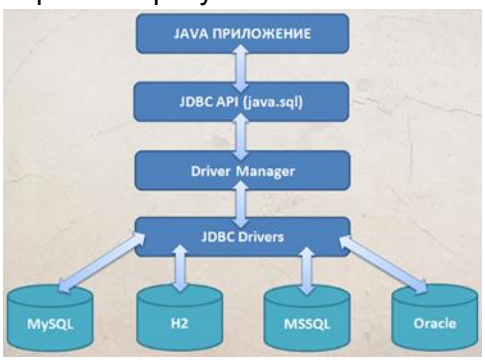
Источники: Java EE 8 Tutorial: Persistence
Java Persistence API
JAVA PERSISTENCE API (JPA) Tutorialspoint PDF

Что такое ORM?

Object Relational Mapping - это концепция/процесс преобразования данных из объектно-ориентированного языка в реляционные БД и наоборот. Например, в Java это делается с помощью рефлексии и JDBC.

JDBC - Java DataBase Connectivity — API для работы с реляционными (зависимыми) БД. Платформенно независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД, реализованный в виде пакета java.sql, входящего в состав Java SE. Предоставляет методы для получения и обновления данных. Не зависит от конкретного типа базы. Библиотека, которая входит в стандартную библиотеу, содержит: набор классов и интерфейсов для работы с БД (для нас разработчиков api) + интерфейсы баз данных.

JDBC реализует механизмы работы подключений к базе данных, создания запросов и обработки результатов.



Наш код –> JDBC –> драйвера разработчиков БД –> БД.

JDBC has 3 entities:

1. Connection (класс). Объект которого отвечает за соединение с базой и режим работы с ней (лекция 3.1.4)
2. Statement (объект для оператора JDBC) используется для отправки SQL-оператора на сервер баз данных. Объект для оператора связан с объектом Connection и является объектом, обрабатывающим взаимодействие между приложением и сервером баз данных. Можно:

- что-то поменять Update statement (create, delete, insert) в базе;
- что-то запросить Query statement (select) из базы;

Для разных задач есть разные виды Statement-ов:

- statement – обычный. передаем в него либо Update либо Query
- PreparedStatement - возможность сделать некий шаблон запроса, подставлять в него к-то значения и использовать его
- CallableStatement - предоставляет возможность вызова хранимой процедуры, расположенной на сервере, из Java™-приложения.

3. ResultSet. Объект с результатом запроса, который вернула база. Внутри него таблица.

Рефлексия - это API, который позволяет:

- получать информацию о переменных, методах внутри класса, о самом классе, его конструкторах, реализованных интерфейсах и т.д.;
- получать новый экземпляр класса;
- получать доступ ко всем переменным и методам, в том числе **приватным;**
- преобразовывать классы одного типа в другой (cast);
- делать все это **во время исполнения программы** (динамически, в **Runtime**).

В Java есть специальный класс по имени Class. Поэтому его и называют классом класса. С помощью него осуществляется работа с рефлексией.

Что такое JPA?

Java Persistence API - это спецификация (стандарт, технология), обеспечивающая объектно-реляционное отображение простых JAVA-объектов (Plain Old Java Object - POJO) и предоставляющая универсальный API для сохранения, получения и управления такими объектами.

Сам JPA не умеет ни сохранять, ни управлять объектами, JPA только определяет правила игры: как должен действовать каждый провайдер (Hibernate, EclipseLink, OJB, Torque и т.д.), реализующий стандарт JPA. Для этого JPA определяет интерфейсы, которые должны быть реализованы провайдерами. Также JPA определяет правила, как должны описываться метаданные отображения и как должны работать провайдеры. Каждый провайдер обязан реализовывать всё из JPA, определяя стандартное получение, сохранение и управление объектами. Помимо этого, провайдеры могут добавлять свои личные классы и интерфейсы, расширяя функционал JPA.

JPA:

- API в пакете javax.persistance (набор интерфейсов EntityManager, Query, EntityTransaction),
- JPQL - объектный язык запросов (запросы выполняются к объектам)
- Metadata (аннотации или xml)

JAVA-код, написанный только с использованием интерфейсов и классов JPA, позволяет разработчику гибко менять одного провайдера на другого. Например, если приложение использует Hibernate как провайдера, то ничего не меняя в коде можно поменять провайдера на любой другой. Но, если мы в коде использовали интерфейсы, классы или аннотации, например, из Hibernate, то поменяв провайдера на EclipseLink, эти интерфейсы, классы или аннотации уже работать не будут.

Что такое Hibernate?

Hibernate - это провайдер, реализующий спецификацию JPA. Hibernate полностью реализует JPA плюс добавляет функционал в виде своих классов и интерфейсов, расширяя свои возможности по работе с сущностями и БД.

2. Что такое EntityManager? Какие функции он выполняет?

Источники:
- Oracle docs - Interface EntityManager
- Java Persistence API - 7
- Hibernate - Persistence Context
- Cuba-platform
- Easyjava
- Jsehelper
- Baeldung - Guide to the Hibernate EntityManager

EntityManager

Это интерфейс JPA, используемый для взаимодействия с персистентным контекстом. EntityManager описывает API для всех основных операций над Entity, а также для получения данных и других сущностей JPA. По сути - главный API для работы с JPA.

Персистентный контекст - это набор экземпляров сущностей, загруженных из БД или только что созданных. Персистентный контекст является своего рода кэшем данных в рамках транзакции - это и есть кэш первого уровня. Внутри контекста персистентности происходит управление экземплярами сущностей и их жизненным циклом. EntityManager автоматически сохраняет в БД все изменения, сделанные в его персистентном контексте, в момент коммита транзакции, либо при явном вызове метода flush().

Один или несколько EntityManager образуют или могут образовать persistence context.

Если проводить аналогию с обычным JDBC, то EntityManagerFactory будет аналогом DataSource, а EntityManager аналогом Connection.

Создание EntityManagerFactory довольно дорогая операция, поэтому обычно её создают один раз и на всё приложение. А чаще всего не создают сами, а делегируют это фреймворку, такому как Spring, например

Интерфейс Session из Hibernate представлен в JPA как раз интерфейсом EntityManager.

JPA	JDBC по аналогии	Hibernate
EntityManagerFactory	DataSource	SessionFactory
EntityManager	Connection	Session
JPQL		HQL

Основные функции EntityManager

1. Операции над Entity: persist (добавление Entity под управление JPA), merge (изменение), remove (удаление), refresh (обновление данных), detach (удаление из-под управления контекста персистентности), lock (блокирование Entity от изменений в других thread).
2. Получение данных: find (поиск и получение Entity), createQuery, createNamedQuery, createNativeQuery, contains, createNamedStoredProcedureQuery, createStoredProcedureQuery.
3. Получение других сущностей JPA: getTransaction, getEntityManagerFactory, getCriteriaBuilder, getMetamodel, getDelegate.

4. Работа с EntityGraph: createEntityGraph, getEntityGraph.
5. Общие операции над EntityManager или всеми Entities: close, isOpen, getProperties, setProperty, clear.

Объекты EntityManager не являются потокобезопасными. Это означает, что каждый поток должен получить свой экземпляр EntityManager, поработать с ним и закрыть его в конце.

3. Каким условиям должен удовлетворять класс, чтобы являться Entity?

Источники:
- Java EE 8 Tutorial - Entities
- Java Persistence API - 2.1-2.3
- Deskbook
- Hibernate - Entity types

Entity

Сущность (entity) - это объект персистентной области. Как правило, сущность представляет таблицу в реляционной базе данных, и каждый экземпляр сущности соответствует строке в этой таблице. Основным программным представлением сущности является класс сущности. Класс сущности может использовать другие классы, которые служат вспомогательными классами или используются для представления состояния сущности (например embedded).

Персистентное состояние сущности представлено персистентными полями или персистентными свойствами.

Персистентное поле - поле сущности, которое отражается в БД в виде столбца таблицы.

Персистентное свойство - это методы, которые аннотированы вместо полей для доступа провайдера к ним (полям).

Эти поля или свойства используют аннотации объектно-реляционного сопоставления (маппинга) для сопоставления сущностей и отношений между ними с реляционными данными в хранилище данных. Примеры аннотаций: @OneToOne, @OneToMany, @ManyToOne, @ManyToMany.

Есть два вида доступа к состоянию сущности:

– Доступ по полю, когда аннотации стоят над полями. В этом случае провайдер, например, Hibernate, обращается к полям класса напрямую, используя Reflection.
– Доступ по свойству, когда аннотации стоят над методами-геттерами. В этом случае провайдер, например, Hibernate, обращается к полям класса через методы.

В JPA принято называть эти персистентные поля и свойства *атрибутами* класса-сущности.

Требования к Entity классу в JPA

1. Entity класс должен быть помечен аннотацией @Entity или описан в XML файле конфигурации JPA.
2. Entity класс должен содержать public или protected конструктор без аргументов (он также может иметь конструкторы с аргументами).
3. Entity класс должен быть классом верхнего уровня (top-level class).
4. Перечисление [enum] или интерфейс [interface] не могут быть определены как сущность [Entity].
5. Entity класс не может быть финальным классом (final class). Entity класс не может содержать финальные поля или методы, если они участвуют в маппинге (persistent final methods or persistent final instance variables).
6. Если объект Entity класса будет передаваться по значению как отделённый от контекста персистентности объект (detached object), например через удаленный интерфейс (through a remote interface), то он также должен реализовывать

интерфейс Serializable (чтобы объекты которые достаются из базы могли сохраняться в кэше).

7. Как обычный так и абстрактный класс может быть Entity. Entities могут наследоваться как от не Entity классов, так и от Entity классов. А не Entity классы могут наследоваться от Entity классов.
8. Поля Entity класса должны быть объявлены private, protected или package-private, быть напрямую доступными только методам самого Entity класса и не должны быть напрямую доступны другим классам, использующим этот Entity. Другие классы должны обращаться только к специальным методам Entity класса, предоставляющим доступ к этим полям (getter/setter-методам или другим методам бизнес-логики в Entity классе).
9. Entity класс должен содержать первичный ключ, то есть атрибут или группу атрибутов, которые уникально определяют запись этого Entity класса в базе данных.

Требования к Entity классу в Hibernate

Hibernate не так строг в своих требованиях. Вот отличия от требований JPA:

- Класс сущности должен иметь конструктор без аргументов, который может быть не только public или protected, но и package visibility (default).
- Класс сущности не обязательно должен быть классом верхнего уровня.
- Технически Hibernate может сохранять финальные классы или классы с финальными методами (getter / setter). Однако, как правило, это не очень хорошая идея, так как это лишит Hibernate возможности генерировать прокси для отложенной загрузки сущности.
- Hibernate не запрещает разработчику приложения открывать прямой доступ к переменным экземпляра и ссылаться на них извне класса сущности. Однако обоснованность такого подхода спорна.

4. Может ли абстрактный класс быть Entity?

Источники: Java Persistence API - 2.11.1

Абстрактный класс может быть Entity классом. Абстрактный Entity класс отличается от обычных Entity классов только тем, что нельзя создать объект этого класса. Имена абстрактных классов могут использоваться в запросах.

Абстрактные Entity классы используются в наследовании, когда их потомки наследуют поля абстрактного класса:

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Employee {
  @Id
  @GeneratedValue
  private long id;
  private String name;
    .............
}

@Entity
@Table(name = "FULL_TIME_EMP")
public class FullTimeEmployee extends Employee {
  private int salary;
    .............
}

@Entity
@Table(name = "PART_TIME_EMP")
public class PartTimeEmployee extends Employee {
  private int hourlyRate;
    .............
}
```

5. Наследование Entity классов.

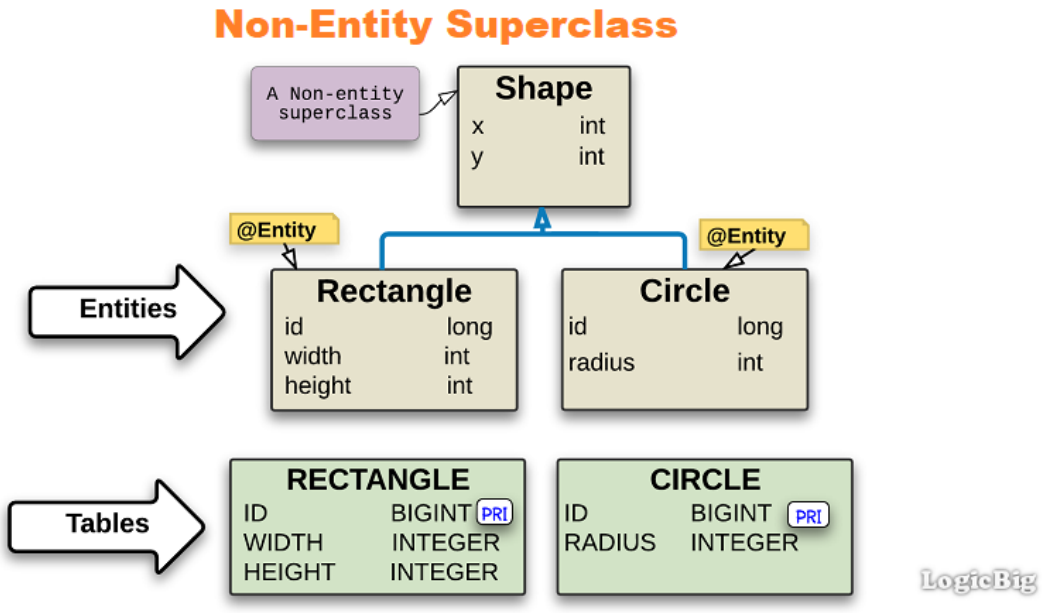
Источники:
- Java EE 8 Tutorial - Entities
- Java EE 8 Tutorial - Non-Entity Superclasses
- LogicBig - JPA - Non-Entity Superclasses

Таблица вариантов наследования Entity классов:

Родитель/Наследник	Entity класс	не Entity класс
Entity класс	+	+
не Entity класс	+	+ (обычное наследование в Java)

Может ли Entity класс наследоваться от не Entity классов (non-entity classes)?

Да, сущности могут наследоваться от не Entity классов, которые, в свою очередь, могут быть как абстрактными, так и обычными. Состояние (поля) не Entity суперкласса не является персистентным, то есть не хранится в БД и не обрабатывается провайдером (Hibernate), поэтому любое такое состояние (поля), унаследованное Entity классом, также не будет отображаться в БД.



Не Entity суперклассы не могут участвовать в операциях EntityManager или Query. Любые маппинги или аннотации отношений в не Entity суперклассах игнорируются.

Может ли Entity класс наследоваться от Entity классов?

Да, может.

Может ли не Entity класс наследоваться от Entity класса?

Да, может.

6. Что такое встраиваемый (Embeddable) класс? Какие требования JPA предъявляет к встраиваемым (Embeddable) классам?

Источники:
- Java Persistence API - 2.5
- Java EE 8 Tutorial - Embeddable Classes
- Hibernate - Embeddable types
- Baeldung - JPA @Embedded And @Embeddable
- LogicBig - Embeddable Classes

Встраиваемый (Embeddable) класс

Это класс, который не используется сам по себе, а только как часть одного или нескольких Entity классов. Hibernate называет эти классы *компонентами*. JPA называет их *встраиваемыми*. В любом случае, концепция одна и та же: композиция значений. Встраиваемый класс помечается аннотацией **@Embeddable**.

Встраиваемый класс может быть встроен в несколько классов-сущностей, но встроенный **объект** с конкретным состоянием принадлежит исключительно владеющей им сущности и не может использоваться одновременно другими сущностями, он не является общим для нескольких сущностей. То есть, если класс Person с полями name и age встроен и в класс Driver, и в класс Baker, то у обоих последних классов появятся оба поля из класса Person. Но если у **объекта** Driver эти поля будут иметь значения “Иван” и “35”, то эти же поля у объекта Baker могут иметь совершенно иные значения, никак не связанные с объектом Driver.

В целом, встраиваемый класс служит для того, чтобы выносить определение общих атрибутов для нескольких сущностей, можно считать что JPA просто встраивает в сущность вместо объекта такого класса те атрибуты, которые он содержит.

Особенности встраиваемых классов

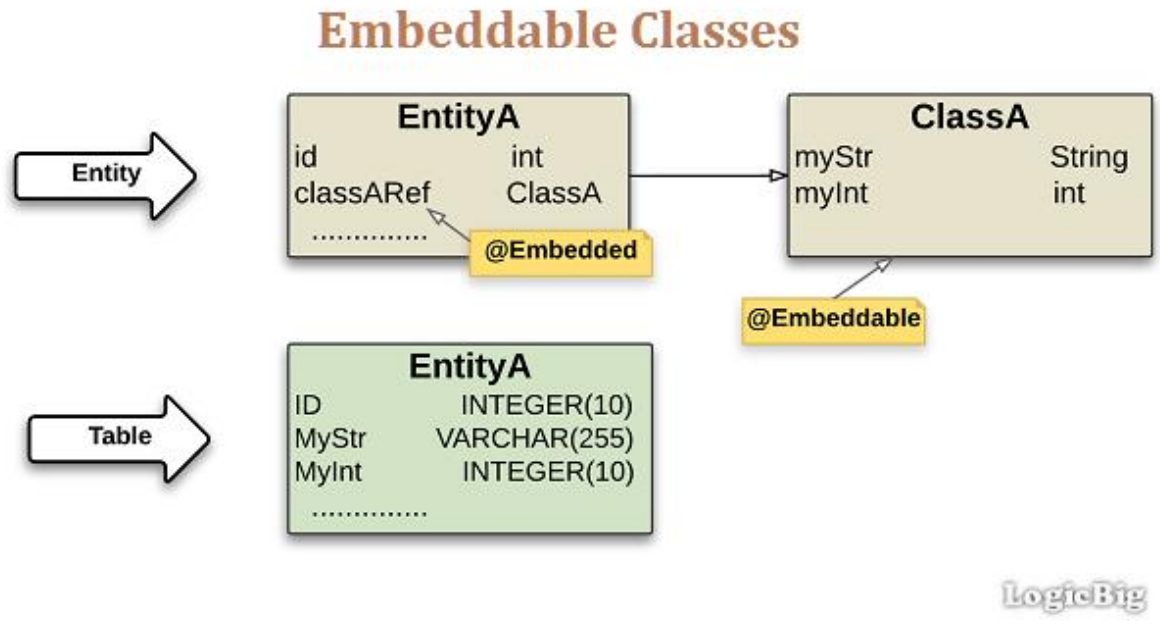
- все поля встраиваемого класса, даже коллекции, станут полями класса, в который происходит встраивание;
- встраиваемые классы могут быть встроены в одну и ту же сущность несколько раз, нужно только поменять имена полей;
- экземпляры встраиваемых классов, в отличие от экземпляров сущностей, не имеют собственного персистентного состояния, вместо этого они существуют только как часть состояния объекта, которому они принадлежат;
- встраиваемые классы могут использовать в качестве полей:
  - базовые типы;
  - коллекции базовых типов (с аннотацией @ElementCollection);
  - другие встраиваемые классы;
  - коллекции других встраиваемых классов (с аннотацией @ElementCollection);
  - сущности;
  - коллекции сущностей;
- сущность может использовать в качестве полей одиночные встраиваемые классы и коллекции встраиваемых классов;
- встраиваемые классы могут использоваться в качестве ключей и значений Map.

Требования к встраиваемым классам

- Должны соответствовать требованиям для сущностей (раздел 2.1 Java Persistence API), за исключением того, что у встраиваемых классов не ставится аннотация @Entity и может отсутствовать первичный ключ (@Id).
- Должны быть аннотированы @Embeddable.

Например, у нас может быть встраиваемый класс ClassA, который представляет собой композицию строкового и числового значений, и эти два поля будут добавлены в класс EntityA:

```
@Entity
public class EntityA {
  @Id
  @GeneratedValue
  private int id;

  @Embedded
  private ClassA classARef;
    .............
}

@Embeddable
public class ClassA {
  private String myStr;
  private int myInt;
    .............
}
```



Так как мы можем встраивать классы в неограниченное количество других классов, то у каждого класса, содержащего встраиваемый класс, мы можем изменить названия полей из встраиваемого класса. Например, у класса Driver поля из встраиваемого класса Person будут изменены с name на driver_name и с age на driver_age:

```
@Embeddable
public class Person {
    private String name;
    private int age;
}

@Entity
public class Driver {

    @Embedded
    @AttributeOverrides({
        @AttributeOverride( name = "name",
                            column = @Column(name = "driver_name")),
        @AttributeOverride( name = "age",
                            column = @Column(name = "driver_age"))
    })
    private Person person;
    ...
}
```

Сущности, которые имеют встраиваемые классы, могут аннотировать поле или свойство аннотацией @Embedded, но не обязаны это делать.

Можно использовать для денормализации БД (ускорений запросов к БД).

7. Что такое Mapped Superclass?

Источники:
Java EE 8 Tutorial - Mapped Superclasses
LogicBig - JPA - Mapped Superclasses
Javastudy
Easyjava - Наследование в JPA

Mapped Superclass (сопоставленный суперкласс) - это класс, от которого наследуются Entity, он может содержать аннотации JPA, однако сам такой класс не является Entity, ему не обязательно выполнять все требования, установленные для Entity (например, он может не содержать первичного ключа). Эти суперклассы чаще всего используются, когда у нас есть общая для нескольких классов сущностей информация о состоянии и отображении, которую можно вынести в Mapped Superclass.

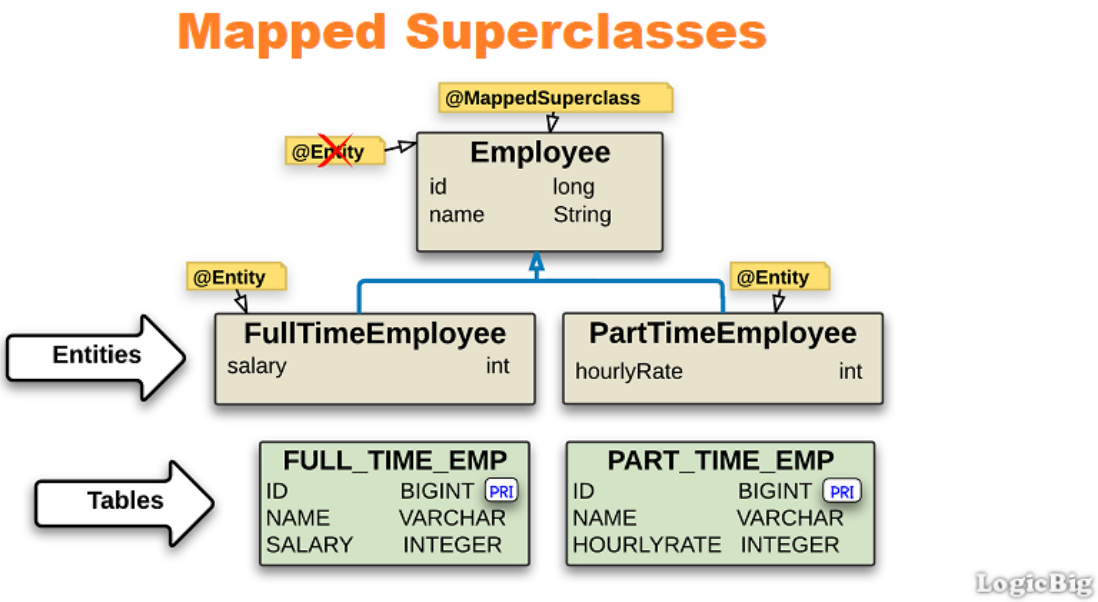
Особенности Mapped Superclass

- Должен быть помечен аннотацией @MappedSuperclass или описан в xml файле.
- Не может использоваться в операциях EntityManager или Query, вместо этого нужно использовать классы-наследники.
- Не может состоять в отношениях с другими сущностями (в сущности нельзя создать поле с типом сопоставленного суперкласса).
- Может быть абстрактным.
- Не имеет своей таблицы в БД.

Для того, чтобы использовать Mapped Superclass, достаточно унаследовать его в классах-потомках:

```
@MappedSuperclass
public class Employee {
  @Id
  @GeneratedValue
  private long id;
  private String name;
    .............
}

@Entity
@Table(name = "FULL_TIME_EMP")
public class FullTimeEmployee extends Employee {
  private int salary;
    .............
}

@Entity
@Table(name = "PART_TIME_EMP")
public class PartTimeEmployee extends Employee {
  private int hourlyRate;
    .............
}
```

В указанном примере кода в БД будут таблицы FULLTIMEEMPLOYEE и PARTTIMEEMPLOYEE, но таблицы EMPLOYEE не будет:



Это похоже на стратегию наследования “Таблица для каждого конкретного класса сущностей”, но в модели данных нет объединения таблиц или наследования. Также тут нет таблицы для Mapped Superclass. Наследование существует только в объектной модели.

Основным недостатком использования сопоставленного суперкласса является то, что полиморфные запросы невозможны, то есть мы не можем загрузить всех наследников Mapped Superclass.

Mapped Superclass vs. Embeddable class

Сходства:

- не являются сущностями и могут иметь все аннотации, кроме @Entity;
- не имеют своих таблиц в БД;
- не могут использоваться в операциях EntityManager или Query.

Различия:

- MappedSuperclass - наследование, Embeddable class - композиция (экземпляр «части» может входить только в одно целое (или никуда не входить));
- поля из Mapped Superclass могут быть у сущности в одном экземпляре, полей из Embeddable class может быть сколько угодно (встроив в сущность Embeddable class несколько раз и поменяв имена полей);
- в сущности нельзя создать поле с типом сопоставленного суперкласса, а с Embeddable можно и нужно.

8. Какие три стратегии маппинга при наследовании сущностей (Entity Inheritance Mapping Strategies) описаны в JPA?

Источники:
- Java EE 8 Tutorial - Entity Inheritance Mapping Strategies
- LogicBig - JPA - Single Table Inheritance Strategy
- LogicBig - JPA - Joined Subclass Inheritance Strategy
- LogicBig - JPA - Table per Class Inheritance Strategy
- Easyjava - Наследование в JPA

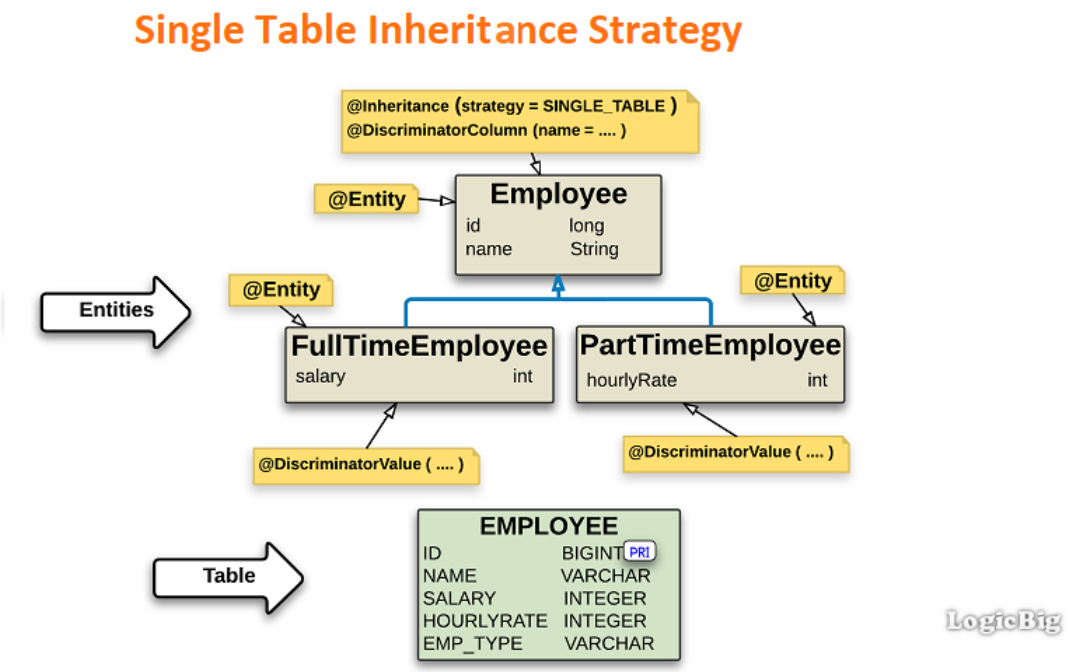
Стратегии наследования нужны для того, чтобы дать понять провайдеру (Hibernate) как ему отображать в БД сущности-наследники. Для этого нам нужно декорировать родительский класс аннотацией @Inheritance и указать один из типов отображения: SINGLE_TABLE, TABLE_PER_CLASS, JOINED.

Следующие три стратегии используются для отображения данных сущности-наследника и родительской сущности:

1. SINGLE_TABLE. Одна таблица на всю иерархию классов.
2. TABLE_PER_CLASS. Таблица для каждого конкретного класса сущностей.
3. JOINED. Стратегия «соединения», при которой поля или свойства, специфичные для подклассов, отображаются в таблицах этих подклассов, а поля или свойства родительского класса отображаются в таблице родительского класса.

Одна таблица на всю иерархию классов (SINGLE TABLE)

Является стратегией по умолчанию и используется, когда аннотация @Inheritance не указана в родительском классе или когда она указана без конкретной стратегии.



Все entity, со всеми наследниками записываются в одну таблицу. Для идентификации типа entity (наследника) определяется специальная колонка “discriminator column”. Например, если есть entity Employee с классами-потомками FullTimeEmployee и PartTimeEmployee, то при такой стратегии все FullTimeEmployee и

PartTimeEmployee записываются в таблицу Employee, и при этом в таблице появляется дополнительная колонка с именем DTYPE, в которой будут записаны значения, определяющие принадлежность к классу. По умолчанию эти значения формируются из имён классов, в нашем случае - либо «FullTimeEmployee» либо «PartTimeEmployee». Но мы можем их поменять в аннотации у каждого класса-наследника: @DiscriminatorValue("F").

Если мы хотим поменять имя колонки, то мы должны указать её новое имя в параметре аннотации у класса-родителя: @DiscriminatorColumn(name=EMP_TYPE).

```
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@Entity
@DiscriminatorColumn(name = "EMP_TYPE")
public class Employee {
  @Id
  @GeneratedValue
  private long id;
  private String name;
}
@Entity
@DiscriminatorValue("F")
public class FullTimeEmployee extends Employee {
  private int salary;
}
@Entity
@DiscriminatorValue("P")
public class PartTimeEmployee extends Employee {
  private int hourlyRate;
}
```

Эта стратегия обеспечивает хорошую поддержку полиморфных отношений между сущностями и запросами, которые охватывают всю иерархию классов сущностей:

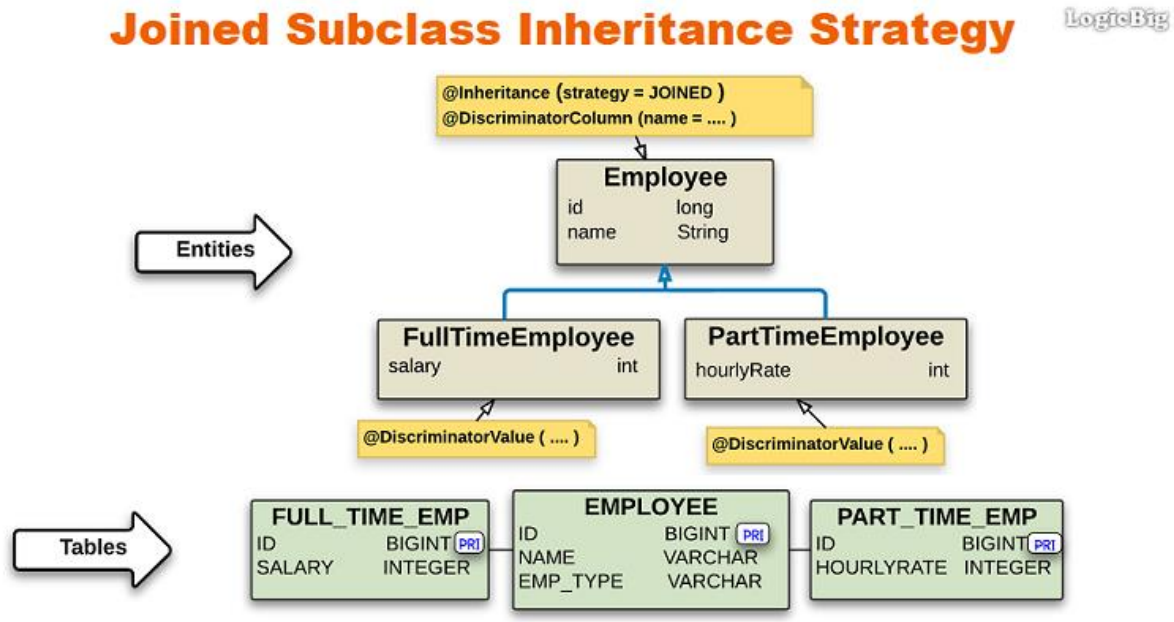
```
-- Persisting entities --
FullTimeEmployee{id=0, name='Sara', salary=100000}
PartTimeEmployee{id=0, name='Tom', hourlyRate='60'}
-- Native queries --
'Select * from Employee'
[F, 1, Sara, null, 100000]
[P, 2, Tom, 60, null]
-- Loading entities --
FullTimeEmployee{id=1, name='Sara', salary=100000}
PartTimeEmployee{id=2, name='Tom', hourlyRate='60'}
```

Минусом стратегии является невозможность применения ограничения NOT NULL для тех колонок таблицы, которые характерны только для классов-наследников.

Стратегия «соединения» (JOINED)

В данной стратегии корневой класс иерархии представлен отдельной таблицей, а каждый класс-наследник имеет свою таблицу, в которой отображены только поля этого класса-наследника. То есть таблица подкласса не содержит столбцы для полей, унаследованных от родительского класса, за исключением поля для первичного ключа @Id, который должен быть определен только в родительской таблице. Столбец

первичного ключа в таблице подкласса служит внешним ключом первичного ключа таблицы суперкласса. Также в таблице родительского класса добавляется столбец DiscriminatorColumn с DiscriminatorValue для определения типа наследника.




```
@Inheritance(strategy = InheritanceType.JOINED)
@Entity
@DiscriminatorColumn(name = "EMP_TYPE") //определение типа наследника
public class Employee {
  @Id
  @GeneratedValue
  private long id;
  private String name;
  ..............
}
@Entity
@DiscriminatorValue("F")
@Table(name = "FULL_TIME_EMP")
public class FullTimeEmployee extends Employee {
  private int salary;
  ..............
}
@Entity
@DiscriminatorValue("P")
@Table(name = "PART_TIME_EMP")
public class PartTimeEmployee extends Employee {
  private int hourlyRate;
  ..............
}
```

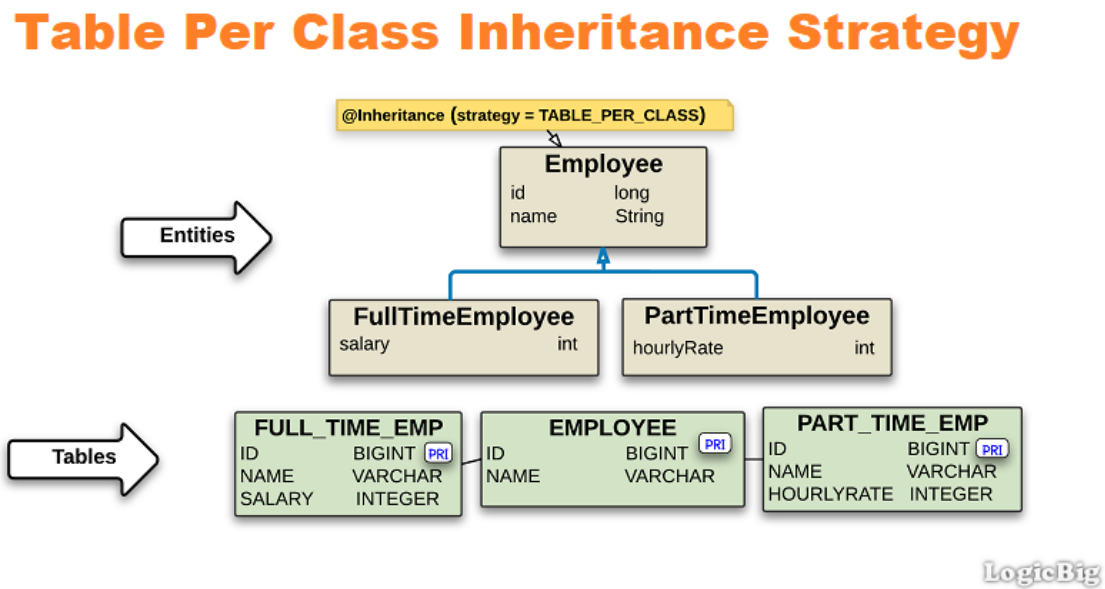
```
-- Persisting entities --
FullTimeEmployee{id=0, name='Sara', salary=100000}
PartTimeEmployee{id=0, name='Robert', hourlyRate='60'}
-- Native queries --
'Select * from Employee'
[F, 1, Sara]
[P, 2, Robert]
'Select * from FULL_TIME_EMP'
[100000, 1]
'Select * from PART_TIME_EMP'
[60, 2]
```

Эта стратегия обеспечивает хорошую поддержку полиморфных отношений, но требует выполнения одной или нескольких операций соединения таблиц при создании экземпляров подклассов сущностей. В глубоких иерархиях классов это может привести к недопустимому снижению производительности. Точно так же запросы, которые покрывают всю иерархию классов, требуют операций соединения между таблицами подклассов, что приводит к снижению производительности:

```
-- Loading entities --
List<Employee> entityAList = em.createQuery("Select t from Employee t")
.getResultList(); // Hibernate makes joins to assemble entities
FullTimeEmployee{id=1, name='Sara', salary=100000}
PartTimeEmployee{id=2, name='Robert', hourlyRate='60'}
```

Таблица для каждого класса сущностей (TABLE PER CLASS)[1]

Каждый класс-наследник имеет свою таблицу. Во всех таблицах подклассов хранятся все поля этого класса плюс те, которые унаследованы от суперкласса.




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

[1] В соответствии со спецификацией Java Persistence API поддержка этой стратегии является необязательной и может поддерживаться не всеми провайдерами.

```
  @Id
  @GeneratedValue
  private long id;
  private String name;
    .............
}
@Entity
@Table(name = "FULL_TIME_EMP")
public class FullTimeEmployee extends Employee {
  private int salary;
    .............
}
@Entity
@Table(name = "PART_TIME_EMP")
public class PartTimeEmployee extends Employee {
  private int hourlyRate;
    .............
}
-- Persisting entities --
FullTimeEmployee{id=0, name='Sara', salary=100000}
PartTimeEmployee{id=0, name='Robert', hourlyRate='60'}
-- Native queries --
'Select * from Employee'
// no data
'Select * from FULL_TIME_EMP'
[1, Sara, 100000]
'Select * from PART_TIME_EMP'
[2, Robert, 60]
-- Loading entities --
List<Employee> entityAList = em.createQuery("Select t from Employee t")
.getResultList(); // Hibernate makes additional sql- or union-queries to get
entities
PartTimeEmployee{id=2, name='Robert', hourlyRate='60'}
FullTimeEmployee{id=1, name='Sara', salary=100000}
```

Минусом является плохая поддержка полиморфизма (polymorphic relationships) и то, что для выборки **всех** классов иерархии потребуется большое количество отдельных sql-запросов для каждой таблицы-наследника или использование UNION-запроса для соединения таблиц всех наследников в одну таблицу.

Также недостатком этой стратегии является повторение одних и тех же атрибутов в таблицах.

При TABLE PER CLASS не работает стратегия генератора первичных ключей IDENTITY, поскольку может быть несколько объектов подкласса, имеющих один и тот же идентификатор, и запрос базового класса приведет к получению объектов с одним и тем же идентификатором (даже если они принадлежат разным типам).

9. Как мапятся Enum`ы?

Источники:
- Hibernate - Mapping enums
- Baeldung - Persisting Enums in JPA
- Vlad Mihalcea - Map an Enum with JPA and Hibernate
- LogicBig - Persisting Java Enum

По порядковым номерам

Если мы сохраняем в БД сущность, у которой есть поле-перечисление (Enum), то в таблице этой сущности создаётся колонка для значений этого перечисления и по умолчанию в ячейки сохраняется порядковый номер этого перечисления (ordinal).

```
public enum MyEnum {
  ConstA, ConstB, ConstC
}
@Entity
public class MyEntity {
  @Id
  private long myId;
  private MyEnum myEnum;

  public MyEntity() {
  }

  public MyEntity(long myId, MyEnum myEnum) {
      this.myId = myId;
      this.myEnum = myEnum;
  }
    ..............
}
```

В JPA типы Enum могут быть помечены аннотацией @Enumerated, которая может принимать в качестве атрибута EnumType.ORDINAL или EnumType.STRING, определяющий, отображается ли перечисление (enum) на столбец с типом Integer или String соответственно.

@Enumerated(EnumType.ORDINAL) - значение по умолчанию, говорит о том, что в базе будут храниться порядковые номера Enum (0, 1, 2…). Проблема с этим типом отображения возникает, когда нам нужно изменить наш Enum. Если мы добавим новое значение в середину или просто изменим порядок перечисления, мы сломаем существующую модель данных. Такие проблемы могут быть трудно уловимыми, и нам придется обновлять все записи базы данных.

По именам

@Enumerated(EnumType.STRING) - означает, что в базе будут храниться имена Enum. С @Enumerated(EnumType.STRING) мы можем безопасно добавлять новые значения перечисления или изменять порядок перечисления. Однако переименование значения enum все равно нарушит работу базы данных. Кроме того, даже несмотря на то, что это представление данных гораздо более читаемо по сравнению с параметром @Enumerated(EnumType.ORDINAL), оно потребляет намного больше места, чем необходимо. Это может оказаться серьезной проблемой, когда нам нужно иметь дело с большим объемом данных.

@PostLoad и @PrePersist

Другой вариант - использование стандартных методов обратного вызова из JPA. Мы можем смапить наши перечисления в БД и обратно в методах с аннотациями @PostLoad и @PrePersist.

Идея состоит в том, чтобы в сущности иметь не только поле с Enum, но и вспомогательное поле. Поле с Enum аннотируем @Transient, а в БД будет храниться значение из вспомогательного поля. Создадим Enum с полем priority. содержащем числовое значение приоритета:

```
public enum Priority {
    LOW(100), MEDIUM(200), HIGH(300);

    private int priority;

    private Priority(int priority) {
        this.priority = priority;
    }

    public int getPriority() {
        return priority;
    }

    public static Priority of(int priority) {
        return Stream.of(Priority.values())
          .filter(p -> p.getPriority() == priority)
          .findFirst()
          .orElseThrow(IllegalArgumentException::new);
    }
}
```

Мы добавили метод Priority.of(), чтобы упростить получение экземпляра Priority на основе его значения int. Теперь, чтобы использовать его в нашем классе Article, нам нужно добавить два атрибута и реализовать методы обратного вызова:

```
@Entity
public class Article {

    @Id
    private int id;
    private String title;
    @Enumerated(EnumType.ORDINAL)
    private Status status;
    @Enumerated(EnumType.STRING)
    private Type type;
    @Basic
    private int priorityValue;
    @Transient
    private Priority priority;

    @PostLoad
    void fillTransient() {
        if (priorityValue > 0) {
            this.priority = Priority.of(priorityValue);
```

```
        }
    }

    @PrePersist
    void fillPersistent() {
        if (priority != null) {
            this.priorityValue = priority.getPriority();
        }
    }
}
```

Несмотря на то, что этот вариант дает нам бОльшую гибкость по сравнению с ранее описанными решениями, он не идеален. Просто кажется неправильным иметь в сущности целых два атрибута, представляющих одно перечисление. Кроме того, если мы используем этот вариант, мы не сможем использовать значение Enum в запросах JPQL.

Converter

В JPA с версии 2.1 можно использовать Converter для конвертации Enum'а в некое его значение для сохранения в БД и получения из БД. Все, что нам нужно сделать, это создать новый класс, который реализует javax.persistence.AttributeConverter и аннотировать его с помощью @Converter.

```
public enum Category {
    SPORT("S"), MUSIC("M"), TECHNOLOGY("T");

    private String code;

    private Category(String code) {
        this.code = code;
    }
    public String getCode() {
        return code;
    }
}

@Entity
public class Article {
    @Id
    private int id;
    private String title;
    @Basic
    private int priorityValue;
    @Transient
    private Priority priority;
    private Category category;
}

@Converter(autoApply = true)
public class CategoryConverter implements AttributeConverter<Category, String> {
     @Override
    public String convertToDatabaseColumn(Category category) {
```

```
        if (category == null) {
            return null;
        }
        return category.getCode();
    }
    @Override
    public Category convertToEntityAttribute(String code) {
        if (code == null) {
            return null;
        }
         return Stream.of(Category.values())
          .filter(c -> c.getCode().equals(code))
          .findFirst()
          .orElseThrow(IllegalArgumentException::new);
    }
}
```

Мы установили @Converter(autoApply=true), чтобы JPA автоматически применял логику преобразования ко всем сопоставленным атрибутам типа Category. В противном случае нам пришлось бы поместить аннотацию @Converter непосредственно над полем Category у каждой сущности, где оно имеется.

В результате в столбце таблицы будут храниться значения: "S", "M" или "T".

Как мы видим, мы можем просто установить наши собственные правила преобразования перечислений в соответствующие значения базы данных, если мы используем интерфейс AttributeConverter. Более того, мы можем безопасно добавлять новые значения enum или изменять существующие, не нарушая уже сохраненные данные. Это решение просто в реализации и устраняет все недостатки с @Enumerated(EnumType.ORDINAL), @Enumerated(EnumType.STRING) и методами обратного вызова.

10. Как мапятся даты (до Java 8 и после)?

Источники:
Hibernate - Mapping Date/Time Values
Baeldung - Hibernate: Mapping Date and Time
Habr - Java и время: часть первая
Habr - Java и время: часть вторая

При работе с датами рекомендуется установить определенный часовой пояс для драйвера JDBC. Таким образом, наше приложение будет независимым от текущего часового пояса системы.

Другой способ - настроить свойство hibernate.jdbc.time_zone в файле свойств Hibernate, который используется для создания фабрики сессий. Таким образом, мы можем указать часовой пояс один раз для всего приложения.

java.sql

Hibernate позволяет отображать различные классы даты/времени из Java в таблицах баз данных. Стандарт SQL определяет три типа даты/времени:

1. DATE - Представляет календарную дату путем хранения лет, месяцев и дней. Эквивалентом JDBC является java.sql.Date.
2. TIME - Представляет время дня и хранит часы, минуты и секунды. Эквивалентом JDBC является java.sql.Time.
3. TIMESTAMP - Хранит как DATE, так и TIME плюс наносекунды. Эквивалентом JDBC является java.sql.Timestamp.

Поскольку эти типы соответствуют SQL, их сопоставление относительно простое. Мы можем использовать аннотацию @Basic или @Column:

```
@Entity
public class TemporalValues {
    @Basic
    private java.sql.Date sqlDate;
    @Basic
    private java.sql.Time sqlTime;
    @Basic
    private java.sql.Timestamp sqlTimestamp;
}
```

Затем мы могли бы установить соответствующие значения следующим образом:

```
temporalValues.setSqlDate(java.sql.Date.valueOf("2017-11-15"));
temporalValues.setSqlTime(java.sql.Time.valueOf("15:30:14"));
temporalValues.setSqlTimestamp(
  java.sql.Timestamp.valueOf("2017-11-15 15:30:14.332"));
```

Обратите внимание, что использование типов java.sql для полей сущностей не всегда может быть хорошим выбором. Эти классы специфичны для JDBC и содержат множество устаревших функций.

Чтобы избежать зависимостей от пакета java.sql, начали использовать классы даты/времени из пакета java.util вместо классов java.sql.Timestamp и java.sql.Time.

java.util

Точность представления времени составляет одну миллисекунду. Для большинства практических задач этого более чем достаточно, но иногда хочется иметь точность повыше.

Поскольку классы в данном API изменяемые (не immutable), использовать их в многопоточной среде нужно с осторожностью. В частности java.util.Date можно признать «эффективно» потоко-безопасным, если вы не вызываете у него устаревшие методы.

java.util.Date

Тип java.util.Date содержит информацию о дате и времени с точностью до миллисекунд. Но так как классы из этого пакета не имели прямого соответствия типам данных SQL, приходилось использовать над полями java.util.Date аннотацию @Temporal, чтобы дать понять SQL, с каким конкретно типом данных она работает. Для этого у аннотации @Temporal нужно было указать параметр TemporalType, который принимал одно из трёх значений: DATE, TIME или TIMESTAMP, что позволяло указать базе данных с какими конкретными типами данных она работает.

```
@Basic
@Temporal(TemporalType.DATE)
private java.util.Date utilDate;

@Basic
@Temporal(TemporalType.TIME)
private java.util.Date utilTime;

@Basic
@Temporal(TemporalType.TIMESTAMP)
private java.util.Date utilTimestamp;
```

Тип java.util.Date имеет точность до миллисекунд, и недостаточно точен для обработки SQL-значения Timestamp, который имеет точность вплоть до наносекунд. Поэтому, когда мы извлекаем сущность из базы данных, неудивительно, что в этом поле мы находим экземпляр java.sql.Timestamp, даже если изначально мы сохранили java.util.Date. Но это не страшно, так как Timestamp наследуется от Date.

java.util.Calendar

Как и в случае java.util.Date, тип java.util.Calendar может быть сопоставлен с различными типами SQL, поэтому мы должны указать их с помощью @Temporal. Разница лишь в том, что Hibernate не поддерживает отображение (маппинг) Calendar на TIME:

```
@Basic
@Temporal(TemporalType.DATE)
private java.util.Calendar calendarDate;

@Basic
@Temporal(TemporalType.TIMESTAMP)
private java.util.Calendar calendarTimestamp;
```

java.time

Начиная с Java 8, доступен новый API даты и времени для работы с временными значениями. Этот API-интерфейс устраняет многие проблемы классов java.util.Date и java.util.Calendar. Все классы в новом API неизменяемые (immutable) и, как следствие, потоко-безопасные. Точность представления времени составляет одну наносекунду, что в миллион раз точнее чем в пакете java.util. Типы данных из пакета java.time напрямую отображаются (маппятся) на соответствующие типы SQL. Поэтому нет необходимости явно указывать аннотацию @Temporal:

1. LocalDate соответствует DATE.
2. LocalTime и OffsetTime соответствуют TIME.
3. Instant, LocalDateTime, OffsetDateTime и ZonedDateTime соответствуют TIMESTAMP.

Это означает, что мы можем пометить эти поля только аннотацией @Basic (или @Column), например:

```
@Basic
private java.time.LocalDate localDate;
@Basic
private java.time.LocalTime localTime;
@Basic
private java.time.OffsetTime offsetTime;
@Basic
private java.time.Instant instant;
@Basic
private java.time.LocalDateTime localDateTime;
@Basic
private java.time.OffsetDateTime offsetDateTime;
@Basic
private java.time.ZonedDateTime zonedDateTime;
```

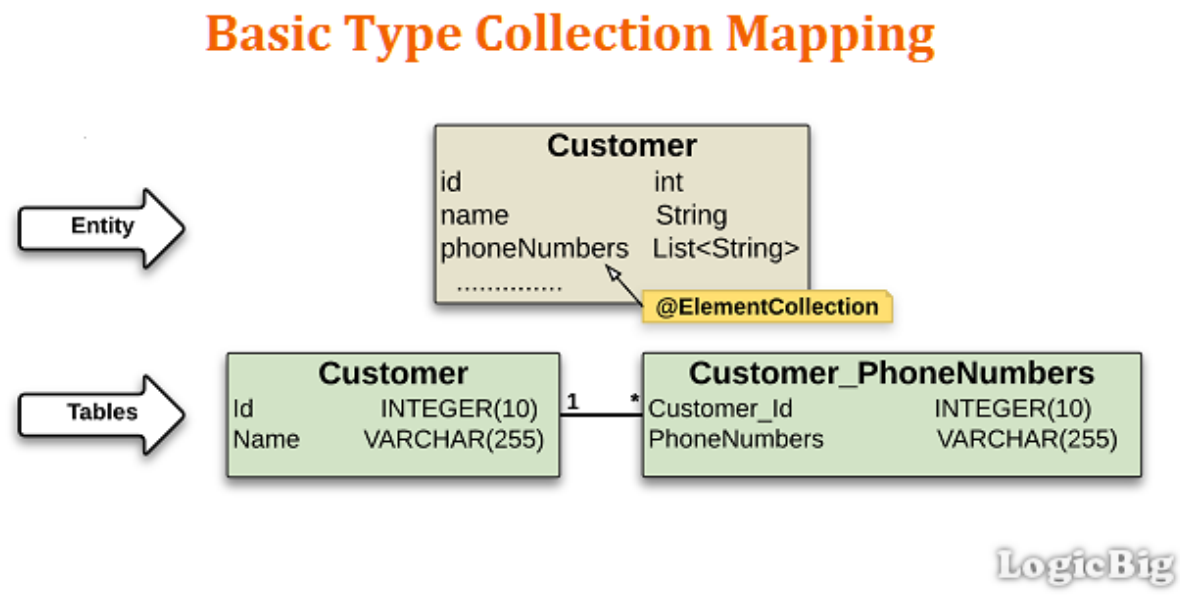
Каждый временной класс в пакете java.time имеет статический метод parse() для анализа предоставленного значения типа String с использованием соответствующего формата. Итак, вот как мы можем установить значения полей сущности:

```
temporalValues.setLocalDate(LocalDate.parse("2017-11-15"));
temporalValues.setLocalTime(LocalTime.parse("15:30:18"));
temporalValues.setOffsetTime(OffsetTime.parse("08:22:12+01:00"));
temporalValues.setInstant(Instant.parse("2017-11-15T08:22:12Z"));
temporalValues.setLocalDateTime(
  LocalDateTime.parse("2017-11-15T08:22:12"));
temporalValues.setOffsetDateTime(
  OffsetDateTime.parse("2017-11-15T08:22:12+01:00"));
temporalValues.setZonedDateTime(
  ZonedDateTime.parse("2017-11-15T08:22:12+01:00[Europe/Paris]"));
```

11. Как сохранять в базе данных коллекции базовых типов?

Источники:
- Java EE 8 Tutorial - Using Collections in Entity Fields and Properties
- Javadoc - @ElementCollection
- LogicBig - JPA - Persisting Collections of basic Java types by using @ElementCollection
- Callicoder - JPA / Hibernate ElementCollection Example
- Baeldung - Persisting Maps with Hibernate
- Hibernate Tips: How to persist a List of Strings as an ElementCollection

Если у нашей сущности есть поле с коллекцией, то мы привыкли ставить над ним аннотации @OneToMany либо @ManyToMany. Но данные аннотации применяются в случае, когда это коллекция других сущностей (entities). Но что, если у нашей сущности коллекция не других сущностей, а базовых или встраиваемых (embeddable) типов, то есть **коллекция элементов**? Для этих случаев в JPA имеется специальная аннотация @ElementCollection, которая указывается в классе сущности над полем коллекции базовых или встраиваемых типов. Все записи коллекции хранятся в отдельной таблице, то есть в итоге получаем две таблицы: одну для сущности, вторую для коллекции элементов. Конфигурация для таблицы коллекции элементов указывается с помощью аннотации @CollectionTable, которая используется для указания имени таблицы коллекции и JoinColumn, который ссылается на первичную таблицу.




```
@Entity
public class Customer {

@Id
  @GeneratedValue
  private int id;

  private String name;
```

```
    @ElementCollection
    private List<String> phoneNumbers;
      .............
}
```

Аннотация @ElementCollection похожа на отношение @OneToMany, за исключением того, что целью являются базовые и встраиваемые типы, а не сущности.

Можно использовать аннотации @AttributeOverrides и @AttributeOverride для настройки отображения в таблице полей базовых или встраиваемых типов.

Коллекции могут иметь тип java.util.Map, которые состоят из ключа и значения. Для этого типа коллекций применяются следующие правила:

1. Ключ или значение Map может быть базовым типом языка программирования Java, встраиваемым классом или сущностью.
2. Если **значение** Map является встраиваемым классом или базовым типом, используйте аннотацию @ElementCollection. Пример.
3. Если **значение** Map является сущностью, используйте аннотацию @OneToMany или @ManyToMany. Пример.
4. Использовать тип Map только на одной стороне двунаправленной связи.

Аннотация @MapKeyColumn позволяет настроить столбец «ключ» в таблице Map. Аннотация @Column позволяет настроить столбец «значение» в таблице Map. Пример.

Использование коллекций элементов имеет один **большой недостаток**: элементы коллекции не имеют идентификатора, и Hibernate не может обращаться индивидуально к каждому элементу коллекции. Когда мы добавляем новый объект в коллекцию или удаляем из коллекции существующий элемент, Hibernate удаляет **все** строки из таблицы элементов и вставляет новые строки по одной для каждого элемента в коллекции. То есть при добавлении одного элемента в коллекцио Hibernate не добавит одну строку в таблицу коллекции, а очистит её и заполнит по новой всеми элементами.

Поэтому коллекции элементов следует использовать только для очень маленьких коллекций, чтобы Hibernate не выполнял слишком много операторов SQL. Во всех других случаях рекомендуется использовать коллекции сущностей с @OneToMany.

12. Какие существуют виды связей?

Источники: Java EE 8 Tutorial - Multiplicity in Entity Relationships

Множественность в отношениях сущностей

Существуют следующие четыре типа связей между сущностями:

1. OneToOne - когда один экземпляр Entity может быть связан не больше чем с одним экземпляром другого Entity.
2. OneToMany - когда один экземпляр Entity может быть связан с несколькими экземплярами других Entity.
3. ManyToOne - обратная связь для OneToMany. Несколько экземпляров Entity могут быть связаны с одним экземпляром другого Entity.
4. ManyToMany - экземпляры Entity могут быть связаны с несколькими экземплярами друг друга.

Направление в отношениях сущностей

Направление отношений может быть как двунаправленным, так и однонаправленным. Двунаправленные отношения имеют как сторону-владельца, так и владеемую сторону. Однонаправленные отношения имеют только сторону-владельца.

Сторона-владелец отношения определяет, как среда выполнения Persistence обновляет отношение в базе данных.

Двунаправленные отношения

В двунаправленном отношении каждая сущность имеет поле, которое ссылается на другую сущность. Через это поле код первой сущности может получить доступ ко второй сущности, находящейся на другой стороне отношений. Если у первой сущности есть поле, ссылающееся на вторую сущность, и наоборот, то в этом случае говорят, что обе сущности знают друг о друге, и что они состоят в двунаправленных отношениях.

Двунаправленные отношения должны следовать следующим правилам:

1. Владеемая сторона в двунаправленных отношениях должна ссылаться на владеющую сторону используя элемент mappedBy аннотаций @OneToOne, @OneToMany, или @ManyToMany. Элемент mappedBy определяет поле в объекте, который является владельцем отношения. Если применить атрибут mappedBy на одной стороне связи, то Hibernate не станет создавать сводную таблицу:

```
@Entity
@Table(name="CART")
public class Cart {
    //...
    @OneToMany(mappedBy="cart")
    private Set<Items> items;
    // getters and setters
}

@Entity
@Table(name="ITEMS")
```

```
public class Items {
    //...
    @ManyToOne
    @JoinColumn(name="cart_id", nullable=false)
    private Cart cart;
    public Items() {}
    // getters and setters
}
```

В данном примере таблица класса Items является владеющей стороной и будет иметь колонку с внешними ключами на таблицу Cart. Таблица класса Cart будет владеемой.

2. Сторона many в отношениях many-to-one **всегда** является владельцем отношений и не может определять элемент mappedBy (такого параметра у аннотации @ManyToOne просто нет).
3. Для двунаправленных отношений one-to-one, сторона-владелец это та сторона, чья таблица имеет столбец с внешним ключом на другую таблицу. Если не указан параметр mappedBy, то колонки с айдишниками появляются у каждой таблицы.
4. Для двунаправленных отношений many-to-many, любая сторона может быть стороной-владельцем.

Однонаправленные отношения

В однонаправленных отношениях только одна сущность имеет поле, которое ссылается на вторую сущность. Вторая сущность (сторона) не имеет поля первой сущности и не знает об отношениях.

Запросы и направление отношений

Язык запросов Java Persistence и запросы API Criteria часто перемещаются между отношениями. Направление отношений определяет, может ли запрос перемещаться от одной сущности к другой. Например в двунаправленных отношениях запрос может перемещаться как от первой сущности ко второй, так и обратно. В однонаправленных отношениях запрос может перемещаться только в одну сторону - от владеющей сущности к владеемой.

13. Что такое “владелец связи”?

Источники: Java EE 8 Tutorial - Direction in Entity Relationships
Baeldung - mappedBy
Doctrine-project

В отношениях между двумя сущностями всегда есть одна владеющая сторона, а владеемой может и не быть, если это однонаправленные отношения.

Владелец связи и владеемый

По сути, у кого есть внешний ключ на другую сущность - тот и владелец связи. То есть, если в таблице одной сущности есть колонка, содержащая внешние ключи от другой сущности, то первая сущность признаётся владельцем связи, вторая сущность - владеемой.

В однонаправленных отношениях сторона, которая имеет поле с типом другой сущности, является владельцем этой связи по умолчанию, например:

```
@Entity
public class LineItem {
    @Id
    private Long id
    @OneToOne
    private Product product;
}
@Entity
public class Product {
    @Id
    private Long id
    private String name;
    private Double price;
}
```

В этом примере владельцем является сущность LineItem, так как она имеет поле типа Product. А Product ничего не знает о LineItem, так как не имеет такого поля. Получается, что в таблице LineItem есть колонка с внешним ключом на таблицу Product, а в таблице Product колонки LineItem нет.

Двунаправленные отношения имеют как сторону-владельца, так и владеемую сторону:

```
@Entity
public class CustomerOrder {
    @Id
    private Long id
    @OneToMany(mappedBy = "customerOrder")
    private Set<LineItem> lineItems = new HashSet<>();
}
@Entity
public class LineItem {
    @Id
    private Long id
    @OneToOne
    private Product product;
```

```
    @ManyToOne
    private CustomerOrder customerOrder;
}
```

В данном примере владельцем связи является сторона LineItem, потому что в таблице LineItem есть колонка с внешними ключами на таблицу CustomerOrder. Да и по правилам Java, сторона @ManyToOne всегда является владельцем связи, а элемент mappedBy определяет поле в объекте, который является владельцем отношения. Тут так и получается: сторона @ManyToOne - LineItem; элемент mappedBy определяет поле "customerOrder" в объекте LineItem.

Хотя интуитивно и кажется, что владельцем связи является сущность CustomerOrder, ведь CustomerOrder - это заказ нашего покупателя, состоящий из позиций (LineItem), и, вроде как, заказ главнее одной позиции, но нет. LineItem в своей таблице владеет ключами от CustomerOrder, поэтому LineItem и является владельцем.

Владельца связи и владеемого легко спутать с родителем отношения и ребенком. В нашем случае CustomerOrder является владеемым и одновременно родителем в отношениях, а LineItem является владеющим и одновременно ребенком в отношениях.

Родительская сущность (таблица)

Это сущность (таблица), на которую ссылается внешний ключ из дочерней сущности (таблицы).

Дочерняя сущность (таблица)

Это сущность (таблица), в которой есть колонка с внешним ключом, ссылающимся на родительскую сущность (таблицу).

14. Что такое каскадные операции?

Источники: Java EE 8 Tutorial - Cascade Operations and Relationships
LogicBig - JPA - Cascading Entities
Stackoverflow: Orphan Removal vs CascadeType.REMOVE

Каскадные операции и отношения

Сущности, между которыми есть отношения, часто зависят от существования друг друга. Например, позиции (LineItem) являются частью заказа (CustomerOrder), и если заказ удален, все позиции также должны быть удалены. Это называется каскадным удалением.

JPA позволяет распространять операции с сущностями (например, persist или remove) на связанные сущности. Это означает, что при включенном каскадировании, если сущность A сохраняется или удаляется, тогда сущность B (связанная с A отношением, например через ManyToOne) также будет сохраняться или удаляться без явных команд сохранения или удаления.

Каскадирования можно добиться, указав у любой из аннотаций @OneToOne, @ManyToOne, @OneToMany, @ManyToMany элемент cascade и присвоив ему одно или несколько значений из перечисления CascadeType (ALL, DETACH, MERGE, PERSIST, REFRESH, REMOVE).

Как правило каскадные операции применяются от родительской сущности к дочерним, но они могут распространяться и в обратном направлении - от дочерней к родительской. Конечно это не всегда нужно, а зачастую совсем не нужно, но главное условие для этого - чтобы между ними было двунаправленное отношение, иначе каскадные операции выполняются только в одном направлении.

Удаление сирот в отношениях (Orphan Removal)

Представим, что у нас есть класс Customer, у которого есть коллекция Order:

```
@Entity
public class Customer {
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Order> orders = new ArrayList<>();
    // other mappings, getters and setters
}
```

Пусть у нас есть один объект Customer - родитель, в коллекции которого есть 4 объекта Order - дети, и мы установили атрибут orphanRemoval = true над этой коллекцией. В нашей базе данных в таблице Customer будет одна строка, а в таблице Order будет четыре строки. Также в таблице Order будет колонка с внешними ключами на таблицу Customer. В каждой из четырех ячеек этой колонки будут ссылки на один и тот же первичный ключ объекта Customer.

Например, мы удалим из коллекции orders один объект Order - любой из четырех, в результате чего у объекта Customer останется три объекта Order:

```
Customer customer = entityManager.find(Customer.class, 1L);
Order order = customer.getOrders().get(0);
customer.getOrders().remove(order);
flushAndClear();
```

После запуска метода flushAndClear() - обновления объекта Customer отправятся в базу данных, и произойдет следующее:

1. Hibernate заметит, что у объекта Customer уже не 4, а 3 связанных дочерних объекта Order;
2. в связи с этим Hibernate найдёт в таблице Order строку с удаленным объектом из коллекции Order;
3. очистит в этой строке ячейку с внешним ключом на Customer;
4. после чего удалит саму эту строку, как осиротевшую (более не ссылающуюся на родителя).

Если не будет атрибута orphanRemoval = true, то пункт 4 не выполнится, и в таблице Order останется сущность Order, не связанная ни с одной сущностью Customer, то есть её ячейка с внешним ключом будет пустой. Такая сущность будет считаться осиротевшей.

15. Какие два типа fetch-стратегии в JPA вы знаете?

Источники: Javadoc - FetchType
Java Persistence API - 2.2
Baeldung - FetchType Strategies

В JPA описаны два типа fetch-стратегии:

1. LAZY — данные поля сущности будут загружены только во время первого обращения к этому полю.
2. EAGER — данные поля будут загружены немедленно вместе с сущностью.

FetchType.EAGER: Hibernate должен сразу загрузить соответствующее аннотированное поле или свойство. Это поведение по умолчанию для полей, аннотированных @Basic, @ManyToOne и @OneToOne (все что быстро).

FetchType.LAZY: Hibernate может загружать данные не сразу, а при первом обращении к ним, но так как это необязательное требование, то Hibernate имеет право изменить это поведение и загружать их сразу. Это поведение по умолчанию для полей, аннотированных @OneToMany, @ManyToMany и @ElementCollection (все что медленно) .

Раньше у Hibernate все поля были LAZY, но в последних версиях - всё как в JPA 2.0.

16. Какие четыре статуса жизненного цикла Entity-объекта (Entity Instance’s Life Cycle) вы можете перечислить?

Источники: Java Persistence API - 3.2
Habr - вопрос 24

Согласно JPA объект сущности может иметь один из четырех статусов жизненного цикла:

1. new - объект создан, не имеет primary key, не является частью контекста персистентности (не управляется JPA);
2. managed - объект создан, имеет primary key, является частью контекста персистентности (управляется JPA);
3. detached - объект создан, имеет primary key, не является (или больше не является) частью контекста персистентности (не управляется JPA);
4. removed - объект создан, является частью контекста персистентности (управляется JPA), будет удален при commit-е транзакции.

17. Как влияет операция persist на объекты Entity каждого статуса?

Источники: Java Persistence API - 3.2
Habr - вопрос 25

persist()

new → managed, объект будет сохранен в базу при commit-е транзакции или в результате flush-операции.

managed → операция игнорируется, однако связанные entity могут поменять статус на managed, если у них есть аннотации каскадных изменений.

removed → managed.

detached → exception сразу или на этапе commit-а транзакции (так как у detached уже есть первичный ключ).

18. Как влияет операция remove на объекты Entity каждого статуса?

Источники: Java Persistence API - 3.2
Habr - вопрос 26

remove()

new → операция игнорируется, однако связанные entity могут поменять статус на removed, если у них есть аннотации каскадных изменений и они имели статус managed.

managed → removed, и запись в базе данных будет удалена при commit-е транзакции (также произойдут операции remove для всех каскадно зависимых объектов).

removed → операция игнорируется.

detached → exception сразу или на этапе commit-а транзакции.

19. Как влияет операция merge на объекты Entity каждого статуса?

Источники: Java Persistence API - 3.2
Habr - вопрос 27

merge()

new → будет создана новая managed entity, в которую будут скопированы данные объекта.

managed → операция игнорируется, однако операция merge сработает на каскадно зависимых entity, если их статус не managed.

removed → exception сразу или на этапе commit-а транзакции.

detached → либо данные будут скопированы в существующую БД managed entity с тем же первичным ключом, либо создана новая managed entity, в которую скопируются данные.

20. Как влияет операция refresh на объекты Entity каждого статуса?

Источники: Java Persistence API - 3.2
Habr - вопрос 28

refresh()
managed → будут восстановлены все изменения из базы данных данного entity, также произойдет refresh всех каскадно зависимых объектов.
new, removed, detached → exception.

21. Как влияет операция detach на объекты Entity каждого статуса?

Источники: Java Persistence API - 3.2
Habr - вопрос 29

detach()
managed, removed → detached.
new, detached → операция игнорируется.

22. Для чего нужна аннотация @Basic?

Источники:
- Java Persistence API - 11.1.6
- Hibernate - Mapping types
- Hibernate - The @Basic annotation
- Baeldung - JPA @Basic Annotation
- Javadoc - @Basic

В широком смысле Hibernate разделяет типы на две группы:
1. Типы значений (Value types).
2. Типы сущностей (Entity types).

Типы сущностей

Сущности из-за своего уникального идентификатора существуют независимо от других объектов, тогда как типы значений нет. Экземпляры сущностей соответствуют строкам в таблице базы данных и различаются между собой благодаря уникальным идентификаторам. Например, две сущности могут иметь абсолютно одинаковые значения полей, но имея разные идентификаторы (первичные ключи) они будут считаться разными, в отличие от POJO, которые при наличии абсолютно одинаковых значений полей будут считаться равными (equals вернет true). Из-за требования к наличию уникального идентификатора, сущности существуют независимо и определяют свой собственный жизненный цикл.

Типы значений

Это данные, которые не определяют свой собственный жизненный цикл. По сути, они принадлежат сущности (entity), которая определяет их жизненный цикл. С другой стороны, всё состояние объекта полностью состоит из типов значений. В свою очередь, типы значений подразделяются на три подкатегории:
1. Базовые типы (Basic types).
2. Встраиваемые типы (Embeddable types).
3. Типы коллекций (Collection types).

Базовый тип значений

Соответствует одному столбцу в БД. Hibernate предоставляет ряд встроенных базовых типов, которые соответствуют естественным отображениям, рекомендованным спецификациями JDBC. Аннотация @Basic может быть применена к полю любого из следующих типов:
1. Примитивы и их обертки.
2. java.lang.String
3. java.math.BigInteger
4. java.math.BigDecimal
5. java.util.Date
6. java.util.Calendar
7. java.sql.Date
8. java.sql.Time
9. java.sql.Timestamp
10. byte[] or Byte[]

11. char[] or Character[]
12. enums
13. любые другие типы, которые реализуют Serializable.

Строго говоря, базовый тип в Hibernate обозначается аннотацией javax.persistence.Basic. Вообще, аннотацию @Basic можно не ставить, как это и происходит по умолчанию. Оба следующих примера в конечном итоге одинаково верны:

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

Аннотация @Basic определяет 2 атрибута:

1. optional - boolean (по умолчанию true) - определяет, может ли значение поля или свойства быть null. Игнорируется для примитивных типов. Но если тип поля не примитивного типа, то при попытке сохранения сущности будет выброшено исключение.
2. fetch - FetchType (по умолчанию EAGER) - определяет, должен ли этот атрибут извлекаться незамедлительно (EAGER) или лениво (LAZY). Однако, это необязательное требование JPA, и провайдерам разрешено незамедлительно загружать данные, даже для которых установлена ленивая загрузка.

Без аннотации @Basic при получении сущности из БД по умолчанию её поля базового типа загружаются принудительно (EAGER) и значения этих полей могут быть null.

23. Для чего нужна аннотация @Column?

Источники:
Java Persistence API - 11.1.9
Hibernate - The @Column annotation
Javadoc - @Column
Stackoverflow: @Column vs @Basic

Аннотация @Column сопоставляет поле класса столбцу таблицы, а её атрибуты определяют поведение в этом столбце, используется для генерации схемы базы данных

@Basic vs @Column:

1. Атрибуты @Basic применяются к сущностям JPA, тогда как атрибуты @Column применяются к столбцам базы данных.
2. @Basic имеет атрибут optional, который говорит о том, может ли поле объекта быть null или нет; с другой стороны атрибут nullable аннотации @Column указывает, может ли соответствующий столбец в таблице быть null.
3. Мы можем использовать @Basic, чтобы указать, что поле должно быть загружено лениво.
4. Аннотация @Column позволяет нам указать имя столбца в таблице и ряд других свойств:
    a. insertable/updatable - можно ли добавлять/изменять данные в колонке, по умолчанию true;
    b. length - длина, для строковых типов данных, по умолчанию 255.

24. Для чего нужна аннотация @Access?

Источники: Java Persistence API - 2.3, 11.1.1
Java EE 8 Tutorial - Persistent Fields and Properties in Entity Classes
Access Strategies in JPA and Hibernate - Which is better, field or property access?

Hibernate или другой провайдер должен каким-то образом получать доступ к полям сущности. Например, при сохранении сущности в базу данных Hibernate должен получить доступ к состоянию сущности, то есть прочитать значения полей сущности, чтобы записать их в соответствующие ячейки таблицы. Аналогично при получении данных из БД и формировании из них объекта сущности, Hibernate должен создать этот самый объект сущности (для этого ему и нужен public или protected конструктор без параметров), а затем записать в поля этого объекта значения, полученные из ячеек БД, тем самым сформировав состояние сущности. Для чтения и записи этих полей Hibernate использует два подхода:

1. Field access (доступ по полям). При таком способе аннотации маппинга (Id, Column, OneToMany, …) размещаются над полями, и Hibernate напрямую работает с полями сущности, читая и записывая их.
2. Property access (доступ по свойствам). При таком способе аннотации размещаются над методами-геттерами, но никак не над сеттерами. Hibernate использует их и сеттеры для чтения и записи полей сущности. Но есть требование - у сущности с property access названия методов должны соответствовать требованиям JavaBeans. Например, если у сущности Customer есть поле с именем firstName, то у этой сущности должны быть определены методы getFirstName и setFirstName для чтения и записи поля firstName.

Совокупность полей и методов (свойств) сущности называется *атрибутами*.

Эти два подхода неявно определяют тип доступа к состоянию конкретной сущности - либо доступ по полям либо доступ по свойствам. Но при неявном определении типа доступа JPA требует, чтобы у всех сущностей в иерархии был единый тип доступа.

По умолчанию тип доступа определяется местом, в котором находится аннотация @Id. Если она будет над полем - это будет AccessType.FIELD, если над геттером - это AccessType.PROPERTY.

Чтобы явно определить тип доступа у сущности, нужно использовать аннотацию **@Access**, которая может быть указана у сущности, Mapped Superclass и Embeddable class, а также над полями или методами.

Аннотация @Access позволяет в иерархии сущностей с одним единым типом доступа безболезненно определить для одной или нескольких сущностей другой тип доступа. То есть в иерархии, где у всех тип доступа, например, field access, можно у какой-нибудь сущности указать тип доступа property access и это не нарушит работу Hibernate.

Если у сущности объявлена аннотация @Access(AccessType.FIELD):

- значит аннотации маппинга нужно размещать над полями;

- есть возможность у любых атрибутов сущности поменять тип доступа на property access, разместив аннотацию @Access(AccessType.PROPERTY) над соответствующими геттерами;
- разместив аннотации маппинга над методами, не имеющими @Access(AccessType.PROPERTY), получим неопределенное поведение.

Если у сущности объявлена аннотация @Access(AccessType.PROPERTY):

- значит аннотации маппинга нужно размещать над геттерами;
- есть возможность у любых атрибутов сущности поменять тип доступа на field access, разместив аннотацию @Access(AccessType.FIELD) над соответствующими полями;
- разместив аннотации маппинга над полями, не имеющими @Access(AccessType.FIELD), получим неопределенное поведение.

Поля, унаследованные от суперкласса, имеют тип доступа этого суперкласса, а не дочерней сущности, даже если они не совпадают.

Когда у одной сущности определены разные типы доступа, то нужно использовать аннотацию @Transient для избежания дублирования маппинга.

25. Для чего нужна аннотация @Cacheable?

Источники:
Java Persistence API - 3.9.1
Hibernate - @Cacheable
Hibernate - Configuring second-level cache mappings
LogicBig - UNSPECIFIED and NONE

@Cacheable - аннотация JPA, используется для указания того, должна ли сущность храниться в кэше второго уровня, в случае, если в файле persistence.xml (или в свойстве javax.persistence.sharedCache.mode конфигурационного файла) для элемента shared-cache-mode установлено одно из значений:

- ENABLE_SELECTIVE: только сущности с аннотацией @Cacheable (равносильно значению по умолчанию @Cacheable(value=true)) будут сохраняться в кэше второго уровня.
- DISABLE_SELECTIVE: все сущности будут сохраняться в кэше второго уровня, за исключением сущностей, помеченных аннотацией @Cacheable(value=false) как некэшируемые.
- ALL: сущности всегда кэшируются, даже если они помечены как некэшируемые.
- NONE: ни одна сущность не кэшируется, даже если помечена как кэшируемая. При данной опции имеет смысл вообще отключить кэш второго уровня.
- UNSPECIFIED: применяются значения по умолчанию для кэша второго уровня, определенные Hibernate. Это эквивалентно тому, что вообще не используется shared-cache-mode, так как Hibernate не включает кэш второго уровня, если используется режим UNSPECIFIED.

Аннотация @Cacheable размещается над классом сущности. Её действие распространяется на эту сущность и её наследников, если они не определили другое поведение.

26. Для чего нужны аннотации @Embedded и @Embeddable?

Источники: Baeldung - JPA @Embedded And @Embeddable

@Embeddable

Аннотация JPA, размещается над классом для указания того, что класс является встраиваемым в другие классы и будет внедрен другими сущностями, то есть поля этого встраиваемого класса будут добавляться к полям других сущностей и будут представлять столбцы в таблице этой сущности. Так, во встраиваемый класс мы можем выделить общие поля для разных сущностей не создавая для него таблицу. Встраиваемый класс сам не является сущностью.

```
@Embeddable
public class ContactPerson {

    private String firstName;
    private String lastName;
    private String phone;
    // standard getters, setters
}
```

@Embedded

Аннотация JPA, используется для размещения над полем в классе-сущности для указания того, что мы внедряем встраиваемый класс.

```
@Entity
public class Company {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;

    private String address;

    private String phone;

    @Embedded
    private ContactPerson contactPerson;

    // standard getters, setters
}
```

Таблица Company будет выглядеть так:

id	name	address	phone	firstName	lastName	phone

27. Как смаппить составной ключ?

Источники:
Java Persistence API - 2.4
Hibernate - Composite identifiers
Baeldung - Composite Primary Keys in JPA
Attacomsian - Composite Primary Key Mapping Example

Составной первичный ключ, также называемый составным ключом, представляет собой комбинацию из двух или более столбцов для формирования первичного ключа таблицы.

В соответствии с JPA, допустимые типы атрибутов для первичного ключа:

1. примитивные типы и их обертки;
2. строки;
3. BigDecimal и BigInteger;
4. java.util.Date и java.sql.Date.

В JPA есть требования к составному ключу:

1. составной ключ должен быть представлен классом первичного ключа, при этом используется одна из двух аннотаций: @IdClass и @EmbeddedId;
2. класс первичного ключа должен быть публичным и иметь публичный конструктор без аргументов;
3. класс первичного ключа должен имплементировать маркерный интерфейс Serializable;
4. класс первичного ключа должен иметь методы equals и hashCode;
5. атрибуты, представляющие поля составного ключа, могут быть базовыми, составными и @ManyToOne, но не могут быть коллекциями или @OneToOne.

Однако, первое правило имеется только в JPA. Hibernate позволяет определять составные идентификаторы без «класса первичного ключа» с помощью нескольких атрибутов с аннотацией @Id.

@IdClass

Допустим, у нас есть таблица с именем Account, и она имеет два столбца - accountNumber и accountType, которые формируют составной ключ. Чтобы обозначить оба этих поля как части составного ключа мы должны создать класс, например, AccountId с этими полями:

```
public class AccountId implements Serializable {
     private String accountNumber;
     private String accountType;
     // default constructor
     public AccountId(String accountNumber, String accountType) {
        this.accountNumber = accountNumber;
        this.accountType = accountType;
     }
     // equals() and hashCode()
}
```

Затем нам нужно аннотировать сущность Account аннотацией @IdClass. Мы также должны объявить поля из класса AccountId в сущности Account **с такими же именами** и аннотировать их с помощью @Id:

```
@Entity
@IdClass(AccountId.class)
public class Account {
    @Id
    private String accountNumber;
    @Id
    private String accountType;
    // other fields, getters and setters
}
```

@EmbeddedId

Является альтернативой аннотации @IdClass.

Рассмотрим другой пример, в котором мы должны сохранить некоторую информацию о книге с заголовком и языком в качестве полей первичного ключа. В этом случае класс первичного ключа, BookId, должен быть аннотирован @Embeddable:

```
@Embeddable
public class BookId implements Serializable {
    private String title;
    private String language;
    // default constructor
    public BookId(String title, String language) {
        this.title = title;
        this.language = language;
    }
    // getters, equals() and hashCode() methods
}
```

Затем нам нужно встроить этот класс в сущность Book, используя @EmbeddedId:

```
@Entity
public class Book {
    @EmbeddedId
    private BookId bookId;
    // constructors, other fields, getters and setters
}
```

@IdClass vs @EmbeddedId

- с @IdClass нам пришлось указывать столбцы дважды - в AccountId и в Account. Но с @EmbeddedId мы этого не сделали;
- JPQL-запросы с @IdClass проще. С @EmbeddedId, чтобы получить доступ к полю, нам нужно из сущности обратиться к встраиваемому классу и потом к его полю:

```
SELECT account.accountNumber FROM Account account // с @IdClass
SELECT book.bookId.title FROM Book book // с @EmbeddedId
```

- @EmbeddedId более подробна, чем @IdClass, поскольку мы можем получить доступ ко всему объекту первичного ключа, используя метод доступа к полю в классе-сущности. Это также дает четкое представление о полях, которые являются частью составного ключа, поскольку все они агрегированы в классе, который доступен только через метод доступа к полям;

- @IdClass может быть предпочтительным выбором по сравнению с @EmbeddedId в ситуациях, когда класс составного первичного ключа поступает из другого модуля или устаревшего кода, а также когда мы не можем его изменить, например, чтобы установить аннотацию @EmbeddedId. Для таких сценариев, где мы не можем изменить класс составного ключа, аннотация @IdClass является единственным выходом;
- если мы собираемся получить доступ к частям составного ключа по отдельности, мы можем использовать @IdClass, но в тех местах, где мы часто используем полный идентификатор в качестве объекта, @EmbeddedId предпочтительнее.

28. Для чего нужна аннотация @ID? Какие GeneratedValue вы знаете?

Источники:
- Java Persistence API - 2.4
- Hibernate - Generated identifier values
- Baeldung - An Overview of Identifiers in Hibernate
- Vlad Mihalcea - How do Identity, Sequence, and Table (sequence-like) generators work in JPA and Hibernate
- How to generate primary keys with JPA and Hibernate
- Stackoverflow - Hibernate disabled insert batching when using an IDENTITY identifier generator

@Id

Аннотация @Id определяет простой (не составной) первичный ключ, состоящий из одного поля. В соответствии с JPA, допустимые типы атрибутов для первичного ключа:

1. примитивные типы и их обертки;
2. строки;
3. BigDecimal и BigInteger;
4. java.util.Date и java.sql.Date.

Стратегии генерации Id

Если мы хотим, чтобы значение первичного ключа генерировалось для нас автоматически, мы можем добавить первичному ключу, отмеченному аннотацией @Id, аннотацию **@GeneratedValue**. Согласно спецификации JPA возможно 4 различных варианта: AUTO, IDENTITY, SEQUENCE, TABLE. Если мы не укажем значение явно, типом генерации по умолчанию будет AUTO. Спецификация JPA строго не определяет поведение этих стратегий.

```
@Id
@GeneratedValue(strategy=TABLE, generator="CUST_GEN")
@Column(name="CUST_ID")
Long id;
```

AUTO

Указывает, что Hibernate должен выбрать подходящую стратегию для конкретной базы данных, учитывая её диалект, так как у разных БД разные способы по умолчанию.

То, как провайдер должен реализовывать тип генерации AUTO, оставлено на усмотрение провайдера. Поведение по умолчанию - исходить из типа поля идентификатора. С версии Hibernate 5.0 для числовых значений генерация основана на SEQUENCE, и, если БД её не поддерживает, то на TABLE.

Пример AUTO, в котором значения первичного ключа будут уникальными на уровне базы данных:

```
@Entity
public class Student {
    @Id
    @GeneratedValue
    private long studentId;
```

```
    // ...
}
```

Интересная особенность, представленная в Hibernate 5, это UUIDGenerator. Чтобы его использовать, всё, что нам нужно сделать, это объявить идентификатор типа UUID с аннотацией @GeneratedValue:

```
@Entity
public class Course {
    @Id
    @GeneratedValue
    private UUID courseId;

    // ...
}
```

Hibernate сгенерирует идентификатор вида «8dd5f315-9788-4d00-87bb-10eed9eff566».

IDENTITY

Указывает, что для генерации значения первичного ключа будет использоваться столбец IDENTITY, имеющийся в базе данных. Значения в столбце автоматически увеличиваются, что позволяет базе данных генерировать новое значение при каждой операции вставки. С точки зрения базы данных это очень эффективно, поскольку столбцы с автоинкрементом хорошо оптимизированы и не требуют каких-либо дополнительных операторов. Процесс инкремента (получения следующего) первичного ключа происходит вне текущей выполняемой транзакции, поэтому откат транзакции может в конечном итоге обнулить уже присвоенные значения (могут возникнуть пропуски значений).

Если мы используем Hibernate, то использование IDENTITY имеет существенный недостаток. Так как Hibernate нужен первичный ключ для работы с managed-объектом в persistence context, а мы не можем узнать значение первичного ключа до выполнения инструкции **INSERT**, то Hibernate должен немедленно выполнить оператор INSERT, чтобы получить этот самый первичный ключ, сгенерированный БД. Только после этого у Hibernate будет возможность работать с сущностью в контексте персистентности, после чего выполнить операцию persist. Но Hibernate, в соответствии со своей идеологией, использует стратегию “транзакционная запись-после” (transactional write-behind), согласно которой он пытается максимально отложить сброс данных в БД из контекста персистентности, чтобы не делать много обращений к БД. Так как поведение при IDENTITY противоречит идеологии и стратегии “транзакционная запись-после”, Hibernate отключает пакетные вставки (batching inserts) для объектов, использующих генератор IDENTITY. Однако, пакетные обновления и удаления (batching updates и batching deletes) всё же поддерживаются.

IDENTITY является самым простым в использовании типом генерации, но не самым лучшим с точки зрения производительности. Как уже упоминалось, стратегия генератора первичных ключей IDENTITY не работает при TABLE PER CLASS, поскольку может быть несколько объектов подкласса, имеющих один и тот же идентификатор, и запрос базового класса приведет к получению объектов с одним и тем же идентификатором (даже если они принадлежат разным типам).

SEQUENCE

Указывает, что для получения значений первичного ключа Hibernate должен использовать имеющиеся в базе данных механизмы генерации последовательных значений (Sequence). Но если наша БД не поддерживает тип SEQUENCE, то Hibernate автоматически переключится на тип TABLE.

SEQUENCE - это объект базы данных, который генерирует инкрементные целые числа при каждом последующем запросе. SEQUENCE намного более гибкий, чем IDENTITY, потому что:

- SEQUENCE не содержит таблиц, и одну и ту же последовательность можно назначить нескольким столбцам или таблицам;
- SEQUENCE может предварительно распределять значения для улучшения производительности;
- SEQUENCE может определять шаг инкремента, что позволяет нам воспользоваться «объединенным» алгоритмом Hilo;
- SEQUENCE не ограничивает пакетные вставки JDBC в Hibernate;
- SEQUENCE не ограничивает модели наследования Hibernate.

При SEQUENCE для получения следующего значения из последовательности базы данных требуются дополнительные операторы **select**, но это не влияет на производительность для большинства приложений. И если нашему приложению необходимо сохранить огромное количество новых сущностей, мы можем использовать некоторые специфичные для Hibernate оптимизации, чтобы уменьшить количество операторов.

Для работы с этой стратегией Hibernate использует свой класс SequenceStyleGenerator.

SEQUENCE - это тип генерации, рекомендуемый документацией Hibernate.

Самый простой способ - просто задать безымянную генерацию последовательности:

```
@Entity(name = "Product")
public static class Product {
	@Id
	@GeneratedValue(
		strategy = GenerationType.SEQUENCE // Имя последовательности не
	// определено, поэтому Hibernate будет использовать последовательность
	// hibernate_sequence для всех сущностей
	)
	private Long id;

	@Column(name = "product_name")
	private String name;

	//Getters and setters are omitted for brevity

}
```

Для всех сущностей с безымянной последовательностью Hibernate будет использовать одну и ту же hibernate_sequence, из которой будет брать для них айдишники.

Используя аннотацию @SequenceGenerator, мы можем указать конкретное имя последовательности для таблицы, а также иные параметры.

Также мы можем настроить под себя несколько разных последовательностей (SEQUENCE-генераторов), указав, например, имя последовательности и начальное значение:

```
@Entity
public class User {
    @Id
    @GeneratedValue(generator = "sequence-generator")
    @GenericGenerator(
      name = "sequence-generator",
      strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",
      parameters = {
        @Parameter(name = "sequence_name", value = "user_sequence"),
        @Parameter(name = "initial_value", value = "4"),
        @Parameter(name = "increment_size", value = "1")
        }
    )
    private long userId;
    // ...
}
```

В этом примере мы установили имя последовательности и начальное значение, что означает, что генерация первичного ключа начнется с 4.

Для каждой последовательности сгенерированные значения являются уникальными.

Так, мы можем назначать разные последовательности разным сущностям и они будут брать айдишники из этой последовательности.

В зависимости от требований приложения, можно иметь один генератор на всё приложение, по генератору на каждую сущность или несколько генераторов, которыми пользуются несколько сущностей.

Например, у нас есть 10 сущностей, для трех из них мы создадим последовательность с именем first_sequence, из которой они будут брать айдишники. Для пяти других сущностей создадим последовательность с именем second_sequence, из которой они будут брать свои айдишники. А для оставшихся двух сущностей можем задать безымянную последовательность, и в этом случае айдишники для них будут браться по умолчанию из hibernate_sequence.

TABLE

В настоящее время GenerationType.TABLE используется редко. Hibernate должен получать первичные ключи для сущностей из специальной создаваемой для этих целей таблицы, способной содержать несколько именованных сегментов значений для любого количества сущностей. Основная идея заключается в том, что данная таблица (например, hibernate_sequence) может содержать несколько сегментов со значениями идентификаторов для разных сущностей. Это требует использования пессимистических блокировок, которые помещают все транзакции по получению идентификаторов в очередь. Разумеется, это замедляет работу приложения.

Третья стратегия, GenerationType.TABLE, не зависит от поддержки конкретной базой данных и хранит счётчики значений в отдельной таблице. С одной стороны это более гибкое и настраиваемое решение, с другой стороны более медленное и требующее большей настройки. Вначале требуется создать *(вручную!)* и проинициализировать *(!)* таблицу для значений ключей.

Затем создать генератор и связать его со идентификатором:
Используя аннотацию @TableGenerator мы можем настроить этот тип генерации:

```
@Entity
public class Department {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE,
      generator = "table-generator")
    @TableGenerator(name = "table-generator",
      table = "dep_ids",
      pkColumnName = "seq_id",
      valueColumnName = "seq_value")
    private long depId;

    // ...
}
```

29. Расскажите про аннотации @JoinColumn, @JoinColumns и @JoinTable. Где и для чего они используются?

Источники: Java Persistence API - 2.4.1.2, 2.4.1.3
Hibernate - @JoinColumn
Baeldung - @JoinColumn
Javadoc - @JoinColumn

@JoinColumn

Аннотация @JoinColumn используется для указания столбца FOREIGN KEY, используемого при установлении связей между сущностями или коллекциями. Мы помним, что только сущность-владелец связи может иметь внешние ключи от другой сущности (владеемой). Однако, мы можем указать аннотацию @JoinColumn как во владеющей таблице, так и во владеемой, но столбец с внешними ключами всё равно появится во владеющей таблице. Особенности использования:

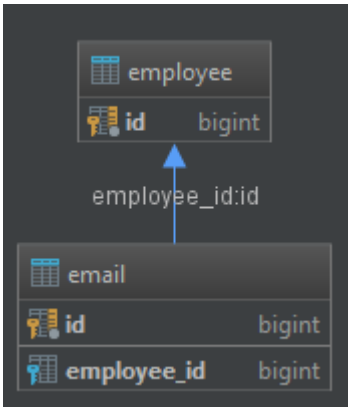
- @OneToOne: означает, что появится столбец addressId в таблице сущности-владельца связи Office, который будет содержать внешний ключ, ссылающийся на первичный ключ владеемой сущности Address.

```
@Entity
public class Office {
    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "addressId")
    private Address address;
}
```

- @OneToMany/@ManyToOne: в данном случае мы можем использовать параметр mappedBy для того, чтобы столбец с внешними ключами находился на владеющей стороне ManyToOne - то есть в таблице Email:

```
@Entity
public class Employee {

    @Id
    private Long id;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "employee")
    private List<Email> emails;
}

@Entity
public class Email {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "employee_id")
    private Employee employee;
}
```

В приведенном выше примере таблица Email (владелец связи) имеет столбец employee_id, в котором хранится значение идентификатора и внешний ключ к таблице Employee.



- Если бы мы не указали mappedBy, то была бы создана **сводная (третья) таблица** с первичными ключами из двух основных таблиц.

@JoinColumns

Аннотация @JoinColumns используется для группировки нескольких аннотаций @JoinColumn, которые используются при установлении связей между сущностями или коллекциями, у которых составной первичный ключ и требуется несколько колонок для указания внешнего ключа.

В каждой аннотации @JoinColumn должны быть указаны элементы name и referencedColumnName:

```
@ManyToOne
@JoinColumns({
    @JoinColumn(name="ADDR_ID", referencedColumnName="ID"),
    @JoinColumn(name="ADDR_ZIP", referencedColumnName="ZIP")
})
public Address getAddress() { return address; }
```

@JoinTable

Аннотация @JoinTable используется для указания связывающей (сводной, третьей) таблицы между двумя другими таблицами.

30. Для чего нужны аннотации @OrderBy и @OrderColumn, чем они отличаются друг от друга?

Источники: Java Persistence API - 11.1.42, 11.1.43
LogicBig - @OrderBy
LogicBig - @OrderColumn

@OrderBy

Аннотация @OrderBy указывает порядок, в соответствии с которым должны располагаться элементы коллекций сущностей, базовых или встраиваемых типов при их извлечении из БД. Эта аннотация может использоваться с аннотациями @ElementCollection, @OneToMany, @ManyToMany.

При использовании с коллекциями **базовых типов**, которые имеют аннотацию @ElementCollection, элементы этой коллекции будут отсортированы в натуральном порядке, по значению базовых типов:

```
@ElementCollection
@OrderBy
private List<String> phoneNumbers;
```

В данном примере коллекция строк phoneNumbers, будет отсортирована в натуральном порядке, по значениям базового типа String.

Если это коллекция **встраиваемых типов** (@Embeddable), то используя точку (".") мы можем сослаться на атрибут внутри встроенного атрибута. Например, следующий код отсортируют адреса по названиям стран в обратном порядке:

```
@Embeddable
public class Address {
    ...
    @Embedded
    private City city
    ....
}
@ElementCollection
@OrderBy("city.country DESC")
private List<Address> addresses;
```

Если это коллекция **сущностей**, то у аннотации @OrderBy можно указать имя поля сущности, по которому сортировать эти самые сущности:

```
@Entity
public class Task {
    ....
    @OneToOne
    private Employee supervisor;
    ...
}
@ManyToMany
@OrderBy("supervisor")
private List<Task> tasks;
```

Если мы не укажем у @OrderBy параметр, то сущности будут упорядочены по первичному ключу.

В случае с сущностями доступ к полю по точке (".") не работает. Попытка использовать вложенное свойство, например @OrderBy ("supervisor.name") повлечет Runtime Exceprtion.

@OrderColumn

Аннотация @OrderColumn создает столбец в таблице, который используется для поддержания постоянного порядка в списке, но этот столбец не считается частью состояния сущности или встраиваемого класса.

Hibernate отвечает за поддержание порядка как в базе данных при помощи столбца, так и при получении сущностей и элементов из БД. Hibernate отвечает за обновление порядка при записи в базу данных, чтобы отразить любое добавление, удаление или иное изменение порядка, влияющее на список в таблице.

Аннотация @OrderColumn может использоваться с аннотациями @ElementCollection, @OneToMany, @ManyToMany - указывается на стороне отношения, ссылающегося на коллекцию, которая должна быть упорядочена. В примере ниже Hibernate добавил в таблицу Employee_PhoneNumbers третий столбец PHONENUMBERS_ORDER, который и является результатом работы @OrderColumn:

```
'Show Columns from Employee_PhoneNumbers'
[EMPLOYEE_ID, BIGINT(19), NO, PRI, NULL]
[PHONENUMBERS, VARCHAR(255), YES, , NULL]
[PHONENUMBERS_ORDER, INTEGER(10), NO, PRI, NULL]
'Select * FROM Employee_PhoneNumbers'
[1, 111-111,111, 0]
[1, 222-222,222, 1]
[2, 333-333-333, 0]
[2, 444-444-444, 1]
[2, 666-666-666, 2]
[3, 555-555-555, 0]
```

@OrderBy vs @OrderColumn

Порядок, указанный в @OrderBy, применяется только в рантайме при выполнении запроса к БД, То есть в контексте персистентности, в то время как при использовании @OrderColumn, порядок сохраняется в отдельном столбце таблицы и поддерживается при каждой вставке/обновлении/удалении элементов.

31. Для чего нужна аннотация @Transient?

Источники: Javabydeveloper - @Transient annotation in Jpa or Hibernate

Аннотация @Transient используется для объявления того, какие поля у сущности, встраиваемого класса или Mapped SuperClass не будут сохранены в базе данных.

Persistent fields (постоянные поля) - это поля, значения которых будут по умолчанию сохранены в БД. Ими являются любые не статические и не финальные поля.

Transient fields (временные поля):

- статические и финальные поля сущностей;
- иные поля, объявленные явно с использованием Java-модификатора transient, либо JPA-аннотации @Transient.

Примеры:

```
static String transient1; // not persistent - ignore because of  static

final String transient2 = "Satish"; // not persistent - ignore because of  final

transient String transient3; // not persistent - ignore because of transient
keyword

@Transient String transient4; // not persistent - ignore because of  @Transient
```

32. Какие шесть режимов блокировок (lock modes) описаны в спецификации JPA (или какие есть значения у enum LockModeType в JPA)?

Источники:
- Java Persistence API - 3.4.4
- Javastudy - Собеседование по Java EE - вопрос 39
- Easyjava - Блокировки в JPA
- LogicBig - OPTIMISTIC_FORCE_INCREMENT
- Baeldung - Pessimistic Locking in JPA

У JPA есть шесть режимов блокировок (lock modes), перечислим их в порядке увеличения надежности (от самого ненадежного и быстрого, до самого надежного и медленного):

1. NONE — без блокировки.
2. OPTIMISTIC (или синоним READ, оставшийся от JPA 1) — оптимистическая блокировка.
3. OPTIMISTIC_FORCE_INCREMENT (или синоним WRITE, оставшийся от JPA 1) — оптимистическая блокировка с принудительным увеличением поля версионности.
4. PESSIMISTIC_READ — пессимистичная блокировка на чтение.
5. PESSIMISTIC_WRITE — пессимистичная блокировка на запись (и чтение).
6. PESSIMISTIC_FORCE_INCREMENT — пессимистичная блокировка на запись (и чтение) с принудительным увеличением поля версионности.

Оптимистичное блокирование

Оптимистичный подход предполагает, что параллельно выполняющиеся транзакции редко обращаются к одним и тем же данным и позволяет им спокойно и свободно выполнять любые чтения и обновления данных. Но при окончании транзакции производится проверка, изменились ли данные в ходе выполнения данной транзакции и, если да, транзакция обрывается и выбрасывается исключение. Оптимистичное блокирование в JPA реализовано путём внедрения в сущность специального поля версии:

```
@Entity
public class Company extends AbstractIdentifiableObject {
    @Version
    private long version;
    @Getter
    @Setter
    private String name;
    @Getter
    @Setter
    @ManyToMany(mappedBy = "workingPlaces")
    private Collection<Person> workers;
}
```

Поле, аннотирование @Version, может быть целочисленным или временнЫм. При завершении транзакции, если сущность была оптимистично заблокирована, будет проверено, не изменилось ли значение @Version кем-либо ещё, после того как данные

были прочитаны, и, если изменилось, будет выкинуто OptimisticLockException. Использование этого поля позволяет отказаться от блокировок на уровне базы данных и сделать всё на уровне JPA, улучшая уровень конкурентности.

JPA поддерживает два типа оптимистичной блокировки:

- LockModeType.OPTIMISTIC — блокировка на чтение, которая работает, как описано выше: если при завершении транзакции кто-то извне изменит поле @Version, то транзакция автоматически будет откачена и будет выброшено OptimisticLockException.
- LockModeType.OPTIMISTIC_FORCE_INCREMENT — блокировка на запись. Ведёт себя как и блокировка на чтение, но при этом увеличивает значение поля @Version.

Обе блокировки ставятся путём вызова метода lock() у EntityManager, в который передаётся сущность, требующая блокировки и уровень блокировки:

```
EntityManager em = entityManagerFactory.createEntityManager();
em.lock(company1, LockModeType.OPTIMISTIC);
em.lock(company2, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
```

Блокировка будет автоматически снята при завершении транзакции, снять её до этого вручную невозможно.

Пессимистичное блокирование

Пессимистичный подход напротив, ориентирован на транзакции, которые постоянно или достаточно часто конкурируют за одни и те же данные и поэтому блокирует доступ к данным превентивно, в тот момент когда читает их. Другие транзакции останавливаются, когда пытаются обратиться к заблокированным данным и ждут снятия блокировки (или кидают исключение). Пессимистичное блокирование выполняется на уровне базы и поэтому не требует вмешательств в код сущности. Так же, как и в случае с оптимистичным блокированием, поддерживаются блокировки чтения и записи:

- LockModeType.PESSIMISTIC_READ — данные блокируются в момент чтения и это гарантирует, что никто в ходе выполнения транзакции не сможет их изменить. Остальные транзакции, тем не менее, смогут параллельно читать эти данные. Использование этой блокировки может вызывать долгое ожидание блокировки или даже выкидывание PessimisticLockException.
- LockModeType.PESSIMISTIC_WRITE — данные блокируются в момент записи и никто с момента захвата блокировки не может в них писать и не может их читать до окончания транзакции, владеющей блокировкой. Использование этой блокировки может вызывать долгое ожидание блокировки.

Кроме того, для сущностей с полем, аннотированным @Version, существует третий вариант пессимистичной блокировки:

- LockModeType.PESSIMISTIC_FORCE_INCREMENT — ведёт себя как LockModeType.PESSIMISTIC_WRITE, но в конце транзакции увеличивает значение поля @Version, даже если фактически сущность не изменилась.

Накладываются пессимистичные блокировки так же как и оптимистичные, вызовом метода lock():

```
em.lock(company1, LockModeType.PESSIMISTIC_READ);
em.lock(company2, LockModeType.PESSIMISTIC_WRITE);
em.lock(company3, LockModeType.PESSIMISTIC_FORCE_INCREMENT);
```

Снимаются они тоже автоматически, по завершению транзакции.

Следующие публичные методы EntityManager-а могут использоваться для наложения блокировок:

```
void lock(Object entity, LockModeType lockMode)
void lock(Object entity, LockModeType lockMode, Map<String, Object>
properties)
<T> T find(Class<T> entityClass, Object primaryKey, LockModeType
lockMode)
<T> T find(Class<T> entityClass, Object primaryKey, LockModeType
lockMode, Map<String, Object> properties)
void refresh(Object entity, LockModeType lockMode)
void refresh(Object entity, LockModeType lockMode, Map<String, Object>
properties)
```

Помимо вышеуказанных методов, в Query API также есть методы для определения блокировок.

33. Какие два вида кэшей (cache) вы знаете в JPA и для чего они нужны?

Источники:
Baeldung - Proxy in Hibernate load() Method
Baeldung - Hibernate Second-Level Cache
Easyjava - Кэширование в Hibernate
Howtodoinjava - First Level Cache

JPA говорит о двух видах кэшей (cache):

1. first-level cache (кэш первого уровня) — кэширует данные одной транзакции;
2. second-level cache (кэш второго уровня) — кэширует данные транзакций от одной фабрики сессий. Провайдер JPA может, но не обязан реализовывать работу с кэшем второго уровня. Такой вид кэша позволяет сэкономить время доступа и улучшить производительность, однако оборотной стороной является возможность получить устаревшие данные.

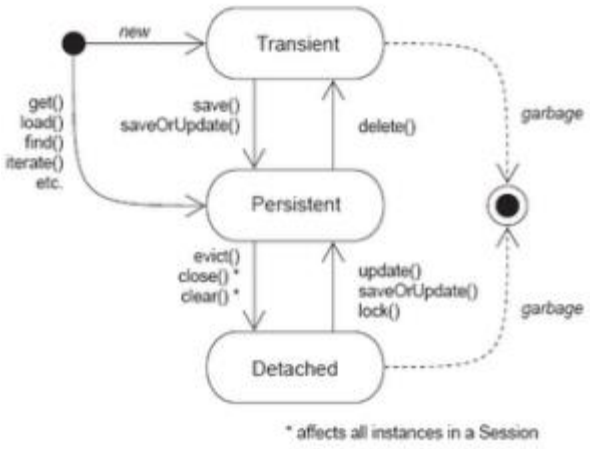
Кэш первого уровня

Кэширование является одним из способов оптимизации работы приложения, ключевой задачей которого является уменьшить количество прямых обращений к базе данных.

Кэш первого уровня – это кэш сессии (Session), который является обязательным, это и есть PersistenceContext. Через него проходят все запросы.

В том случае, если мы выполняем несколько обновлений объекта, Hibernate старается отсрочить (насколько это возможно) обновление этого объекта для того, чтобы сократить количество выполненных запросов в БД. Например, при пяти истребованиях одного и того же объекта из БД в рамках одного persistence context, запрос в БД будет выполнен один раз, а остальные четыре загрузки будут выполнены из кэша. Если мы закроем сессию, то все объекты, находящиеся в кэше, теряются, а далее – либо сохраняются в БД, либо обновляются.

Интересно поведение кэша первого уровня при использовании ленивой загрузки. При загрузке объекта методом load() или объекта с лениво загружаемыми полями, лениво загружаемые данные в кэш не попадут, а вместо этих данных Hibernate создаст объект Proxy. Однако, как только мы обратимся к этому прокси-объекту в рамках этого же открытого контекста персистентности, Hibernate всё-таки выполнит запрос в базу и данные будут загружены и в объект и в кэш. А вот следующая попытка лениво загрузить объект приведёт к тому, что объект сразу вернут из кэша уже полностью загруженным, без обращения в БД.



Особенности кэша первого уровня:

- включен по умолчанию, его нельзя отключить;
- связан с сессией (контекстом персистентности), то есть разные сессии видят только объекты из своего кэша, и не видят объекты, находящиеся в кэшах других сессий;
- при закрытии сессии контекст персистентности очищается - кэшированные объекты, находившиеся в нем, удаляются;
- при первом запросе сущности из БД, она загружается в кэш, связанный с этой сессией;
- если в рамках этой же сессии мы снова запросим эту же сущность из БД, то она будет загружена из кэша, и никакого второго SQL-запроса в БД сделано не будет;
- сущность можно удалить из кэша сессии методом evict(), после чего следующая попытка получить эту же сущность повлечет обращение к базе данных;
- метод clear() очищает весь кэш сессии.

Кэш второго уровня

Если кэш первого уровня привязан к объекту сессии, то кэш второго уровня привязан к объекту-фабрике сессий (Session Factory object) и, следовательно, кэш второго уровня доступен одновременно в нескольких сессиях или контекстах персистентности. Кэш второго уровня требует некоторой настройки и поэтому не включен по умолчанию. Настройка кэша заключается в конфигурировании реализации кэша и разрешения сущностям быть закэшированными.

Hibernate не реализует сам никакого in-memory cache, а использует существующие реализации кэшей.

34. Как работать с кэшем 2 уровня?

Источники:
Java Persistence API - 3.9
Hibernate - Caching
Baeldung - Hibernate Second-Level Cache
Easyjava - Кэширование в Hibernate
Habr - Hibernate cache
Stackoverflow - How to use JPA2's @Cacheable instead of Hibernate's @Cache

Если кэш первого уровня привязан к объекту сессии, то кэш второго уровня привязан к объекту-фабрике сессий (Session Factory object). Что как бы подразумевает, что видимость этого кэша гораздо шире кэша первого уровня.

Чтение из кэша второго уровня происходит только в том случае, если нужный объект не был найден в кэше первого уровня.

Hibernate поставляется со встроенной поддержкой стандарта кэширования Java JCache, а также двух популярных библиотек кэширования: Ehcache и Infinispan.

Shared Cache Mode

Будут ли в нашем приложении кэшироваться сущности и связанные с ними состояния, определяется значением элемента shared-cache-mode файла persistence.xml (или в свойстве javax.persistence.sharedCache.mode конфигурационного файла). Если в файле для элемента shared-cache-mode установлено значение:

- ENABLE_SELECTIVE (дефолтное и рекомендуемое значение): только сущности с аннотацией @Cacheable (равносильно значению по умолчанию @Cacheable(value=true)) будут сохраняться в кэше второго уровня.
- DISABLE_SELECTIVE: все сущности будут сохраняться в кэше второго уровня, за исключением сущностей, помеченных аннотацией @Cacheable(value=false) как некэшируемые.
- ALL: сущности всегда кэшируются, даже если они помечены как некэшируемые.
- NONE: ни одна сущность не кэшируется, даже если помечена как кэшируемая. При данной опции имеет смысл вообще отключить кэш второго уровня.
- UNSPECIFIED: применяются значения по умолчанию для кэша второго уровня, определенные Hibernate. Это эквивалентно тому, что вообще не используется shared-cache-mode, так как Hibernate не включает кэш второго уровня, если используется режим UNSPECIFIED.

В Hibernate кэширование второго уровня реализовано в виде абстракции, то есть мы должны предоставить любую её реализацию, вот несколько провайдеров: Ehcache, OSCache, SwarmCache, JBoss TreeCache. Для Hibernate требуется только реализация интерфейса org.hibernate.cache.spi.RegionFactory, который инкапсулирует все детали, относящиеся к конкретным провайдерам. По сути, RegionFactory действует как мост между Hibernate и поставщиками кэша. В примерах будем использовать Ehcache. Что нужно сделать:

- добавить мавен-зависимость кэш-провайдера нужной версии:

```
<dependency>
```

```
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>5.2.2.Final</version>
</dependency>
```

- включить кэш второго уровня и определить конкретного провайдера:

```
hibernate.cache.use_second_level_cache=true
hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCacheRegionFactory
```

- установить у нужных сущностей JPA-аннотацию @Cacheable, обозначающую, что сущность нужно кэшировать, и Hibernate-аннотацию @Cache, настраивающую детали кэширования, у которой в качестве параметра указать *стратегию параллельного доступа* (о которой говорится далее), например так:

```
@Entity
@Table(name = "shared_doc")
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class SharedDoc{
    private Set<User> users;
}
```

- не обязательно устанавливать у сущностей JPA-аннотацию @Cacheable, если работаем с Hibernate напрямую, не через JPA.
- чтобы кэш не “съел” всю доступную память, можно, например, ограничивать количество каждого типа сущностей, хранимых в кэше:

```
<ehcache>
   <cache name="com.baeldung.persistence.model.Foo" maxElementsInMemory="1000"/>
</ehcache>
```

Стратегия параллельного доступа к объектам

Проблема заключается в том, что кэш второго уровня доступен из нескольких сессий сразу и несколько потоков программы могут одновременно в разных транзакциях работать с одним и тем же объектом. Следовательно надо как-то обеспечивать их одинаковым представлением этого объекта. В Hibernate существует четыре стратегии одновременного доступа к объектам в кэше:

- READ_ONLY: Используется только для сущностей, которые никогда не изменяются (будет выброшено исключение, если попытаться обновить такую сущность). Очень просто и производительно. Подходит для некоторых статических данных, которые не меняются.
- NONSTRICT_READ_WRITE: Кэш обновляется после совершения транзакции, которая изменила данные в БД и закоммитила их. Таким образом, строгая согласованность не гарантируется, и существует небольшое временное окно между обновлением данных в БД и обновлением тех же данных в кэше, во время которого параллельная транзакция может получить из кэша устаревшие данные.
- READ_WRITE: Эта стратегия гарантирует строгую согласованность, которую она достигает, используя «мягкие» блокировки: когда обновляется кэшированная сущность, на нее накладывается мягкая блокировка, которая снимается после коммита транзакции. Все параллельные транзакции, которые пытаются получить доступ к записям в кэше с наложенной мягкой блокировкой, не смогут их

прочитать или записать и отправят запрос в БД. Ehcache использует эту стратегию по умолчанию.

- TRANSACTIONAL: полноценное разделение транзакций. Каждая сессия и каждая транзакция видят объекты, как если бы только они с ним работали последовательно одна транзакция за другой. Плата за это — блокировки и потеря производительности.

Представление объектов в кэше

Еще одна важная деталь про кэш второго уровня о которой стоило бы упомянуть — Hibernate не хранит сами объекты Ваших классов. Он хранит информацию в виде массивов строк, чисел и т.д. Что очень разумно, учитывая сколько лишней памяти занимает каждый объект. Идентификатор объекта выступает указателем на эту информацию. Концептуально это нечто вроде Map, в которой id объекта — ключ, а массивы данных — значения полей. Приблизительно это можно представить себе так:

```
1 -> { "Pupkin", 1, null , {1,2,5} }
```

Помимо вышесказанного, следует помнить — зависимости Вашего класса по умолчанию также не кэшируются. Например, рассмотрим класс SharedDoc, который кэшируется:

```
@Entity
@Table(name = "shared_doc")
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class SharedDoc{
    private Set<User> users;
}
```

В примере выше при выборке сущности SharedDoc из кэша, коллекция users будет доставаться из БД, а не из кэша второго уровня. Если мы хотим также кэшировать и зависимости, то над полями тоже нужно разместить аннотации @Cacheable и @Cache:

```
@Entity
@Table(name = "shared_doc")
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class SharedDoc{
    @Cacheable
    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
    private Set<User> users;
}
```

Однако, при кэшировании коллекций, содержащих другие сущности, будут закэшированы только их первичные ключи. Если это коллекция базовых типов, то будут храниться сами значения базовых типов.

@Cache

Это аннотация Hibernate, настраивающая тонкости кэширования объекта в кэше второго уровня Hibernate. @Cache принимает три параметра:

- include - имеет по умолчанию значение all и означающий кэширование всего объекта. Второе возможное значение - non-lazy, запрещает кэширование лениво

загружаемых объектов. Кэш первого уровня не обращает внимания на эту директиву и всегда кэширует лениво загружаемые объекты.

- region - позволяет задать имя региона кэша для хранения сущности. Регион можно представить как разные области кэша, имеющие разные настройки на уровне реализации кэша. Например, можно было бы создать в конфигурации ehcache два региона, один с краткосрочным хранением объектов, другой с долгосрочным и отправлять часто изменяющиеся объекты в первый регион, а все остальные - во второй. Ehcache по умолчанию создает регион для каждой сущности с именем класса этой сущности, соответственно в этом регионе хранятся только эти сущности. К примеру, экземпляры Foo хранятся в Ehcache в кэше с именем “com.baeldung.hibernate.cache.model.Foo”.
- usage - задаёт стратегию одновременного доступа к объектам.

Кэш запросов (Query Cache)

Результаты HQL-запросов также могут быть кэшированы. Это полезно, если мы часто выполняем запрос к объектам, которые редко меняются. Чтобы включить кэш запросов, установите для свойства hibernate.cache.use_query_cache значение true:

```
hibernate.cache.use_query_cache=true
```

Затем для **каждого** запроса мы должны явно указать, что запрос кэшируется через подсказку в запросе setHint("org.hibernate.cacheable", true):

```
entityManager.createQuery("select f from Foo f")
  .setHint("org.hibernate.cacheable", true)
  .getResultList();
```

Кэш запросов похож на кэш второго уровня. Но в отличии от него, ключом к данным кэша выступает не идентификатор объекта, а совокупность параметров запроса. А сами данные — это идентификаторы объектов, соответствующих критериям запроса, а не значения полей сущностей. И, чтобы получить закэшированный объект, мы должны по данному идентификатору его найти в этом же кэше. Именно поэтому кэш запросов рационально использовать с кэшем второго уровня.

У кэша запросов есть и своя цена — Hibernate будет вынужден отслеживать сущности закешированные с определённым запросом и выкидывать запрос из кэша, если кто-то поменяет значение сущности. То есть для кэша запросов стратегия параллельного доступа всегда read-only.

35. Что такое JPQL/HQL и чем он отличается от SQL?

Источники: Javastudy - Собеседование по Java EE - вопросы 45, 46
Hibernate - HQL and JPQL
Easyjava - HQL

Hibernate Query Language (HQL) и **Java Persistence Query Language (JPQL)** - оба являются объектно-ориентированными языками запросов, схожими по природе с SQL. JPQL - это подмножество HQL. JPQL-запрос всегда является допустимым HQL-запросом, однако обратное неверно.

Java Persistence query language (JPQL)

Это язык запросов, практически такой же как SQL, однако, вместо имен и колонок таблиц базы данных, он использует имена классов Entity и их атрибуты. В качестве параметров запросов также используются типы данных атрибутов Entity, а не полей баз данных. В отличии от SQL в JPQL есть автоматический полиморфизм. Также в JPQL используются функции, которых нет в SQL: такие как KEY (ключ Map'ы), VALUE (значение Map'ы), TREAT (для приведения суперкласса к его объекту-наследнику, downcasting), ENTRY и т.п.

Полиморфные запросы

В отличии от SQL в запросах JPQL есть автоматический полиморфизм, то есть каждый запрос к Entity возвращает не только объекты этого Entity, но также объекты всех его классов-потомков, независимо от стратегии наследования (например, запрос select * from Animal, вернет не только объекты Animal, но и объекты классов Cat и Dog, которые унаследованы от Animal). Чтобы исключить такое поведение используется функция TYPE в where условии (например select * from Animal a where TYPE(a) IN (Animal, Cat) уже не вернет объекты класса Dog).

В JPA запрос представлен в виде javax.persistence.Query или javax.persistence.TypedQuery, полученных из EntityManager. Для создания Query или TypedQuery необходимо использовать метод EntityManager#createQuery. Для именованных запросов необходим метод EntityManager#createNamedQuery.

Запросы, как обычные, так и именованные, формируются из EntityManager:

```
Query query = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name"
);
TypedQuery<Person> typedQuery = entityManager.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name", Person.class
);
@NamedQuery(
    name = "get_person_by_name",
    query = "select p from Person p where name = :name"
)
```

```
Query query = entityManager.createNamedQuery( "get_person_by_name" );

TypedQuery<Person> typedQuery = entityManager.createNamedQuery(
    "get_person_by_name", Person.class
);
```

Hibernate Query Language (HQL)

В Hibernate HQL-запрос представлен org.hibernate.query.Query, полученный из Session. Если HQL является именованным запросом, то будет использоваться Session#getNamedQuery, в противном случае требуется Session#createQuery. HQL пердоставляет дополнительные возможности по сравнению с JPQL.

```
org.hibernate.query.Query query = session.createQuery(
    "select p " +
    "from Person p " +
    "where p.name like :name"
);
org.hibernate.query.Query query = session.getNamedQuery( "get_person_by_name" );
```

36. Что такое Criteria API и для чего он используется?

Источники:
- Javastudy - Собеседование по Java EE - вопрос 47
- Easyjava - JPA Criteria API
- Easyjava - Hibernate Criteria API
- Javastudy - JPA Criteria API
- LogicBig - Getting started with JPA Criteria API
- Objectdb - JPA Criteria API vs JPQL
- IBM - JPA Criteria API

Начиная с версии 5.0 собственный Hibernate Criteria API признан устаревшим и не развивается. Вместо него рекомендуется использовать JPA Criteria API.

Начиная с версии 5.2 Hibernate Criteria API объявлен deprecated и не рекомендуется к использованию.

Hibernate Criteria API

Это тоже язык запросов, аналогичный JPQL (Java Persistence query language), однако запросы основаны на методах и объектах. Hibernate Criteria API является более объектно-ориентированным для запросов, которые получают результат из базы данных. Для операций update, delete или других DDL манипуляций использовать Criteria API нельзя. Критерии используются **только для выборки из базы данных** в более объектно-ориентированном стиле. Используется для динамических запросов. Запросы выглядят так:

```
session.createCriteria(Person.class)
  .setMaxResults(10)
  .list()
  .forEach(System.out::println);
```

Запрос выше полностью аналогичен запросу HQL "from Person". С Criteria также работают и все те вещи, которые работают и с Query: пейджинг, таймауты и т.д.

Разумеется, в Criteria запросах можно и нужно накладывать условия, по которым объекты будут отбираться:

```
session.createCriteria(Person.class)
  .add(Restrictions.eq("lastName", "Testoff"))
  .list()
  .forEach(System.out::println);
```

JPA Criteria API

Criteria API - это актуальный API, используемый для определения запросов для сущностей. Это альтернативный способ определения JPQL-запроса. Эти запросы типобезопасны, переносимы и легко меняются путем изменения синтаксиса.

Основные преимущества JPA Criteria API:

- ошибки могут быть обнаружены во время компиляции;
- позволяет динамически формировать запросы на этапе выполнения приложения.

Запросы на основе строк JPQL и запросы на основе критериев JPA одинаковы по производительности и эффективности.

Для простых статических запросов предпочтительнее использовать строковые запросы JPQL (например, в виде именованных запросов). Для динамических запросов, которые создаются во время выполнения - JPA Criteria API может быть предпочтительней. Например, построение динамического запроса на основе полей, которые пользователь заполняет в рантайме в форме, которая содержит много необязательных полей. Ожидается, что построение этого запроса будет более ясным и понятным при использовании JPA Criteria API, поскольку устраняет необходимость в создании запроса с использованием многих операций конкатенации строк.

Пример использования JPA Criteria API:

```
EntityManager em = entityManagerFactory.createEntityManager();
em.getTransaction().begin();
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Person> personCriteria = cb.createQuery(Person.class);
Root<Person> personRoot = personCriteria.from(Person.class);
personCriteria.select(personRoot);
em.createQuery(personCriteria)
  .getResultList()
  .forEach(System.out::println);
```

Алгоритм: создаём EntityManager, открываем транзакцию и создаём CriteriaBuilder, который будет строить объекты запросов. С помощью CriteriaBuilder создаём CriteriaQuery, который параметризуется типом, который этот запрос возвращает. Затем создаём корневой объект, от которого производится обход дерева свойств при накладывании ограничений или указании, что выбирать. Последним шагом говорится, что же мы хотим выбрать и, наконец, запрос отправляется в EntityManager, где и выполняется как обычно. Построенный выше весьма многословный пример эквивалентен JPQL запросу «from Person».

Все шаги, перечисленные выше, являются обязательными для создания запроса с помощью Criteria API. Важно понимать, что корневой объект указывает JPQL, откуда будут браться данные, а CriteriaQuery указывает тип возвращаемых данных. И типы Root и CriteriaQuery могут отличаться:

```
CriteriaQuery<Passport> passportCriteria = cb.createQuery(Passport.class);
Root<Person> personPassportRoot = passportCriteria.from(Person.class);
passportCriteria.select(personPassportRoot.get("passport"));
em.createQuery(passportCriteria)
  .getResultList()
  .forEach(System.out::println);
```

Этот запрос аналогичен JPQL запросу «select passport from Person» и показывает, что класс, из которого запрашиваются данные и класс, который вернёт запрос, могут быть разными.

Metamodel и типобезопасность.

Все примеры выше решают проблему с программным созданием запросов, но всё ещё бессильны перед изменениями сущностей. В самом деле, изменив в сущности Person поле workingPlaces на jobs - развалится этот запрос:

```
CriteriaQuery<Person> personWorkCriteria = cb.createQuery(Person.class);
Root<Person> personWorkRoot = personWorkCriteria.from(Person.class);
```

```
Join<Person, Company> company = personWorkRoot.join("workingPlaces");
personWorkCriteria.select(personWorkRoot);
personWorkCriteria.where(cb.equal(company.get("name"), "Acme Ltd"));
em.createQuery(personWorkCriteria)
  .getResultList()
  .forEach(System.out::println);
```

И мы не узнаем, что он развалится, пока не попробуем его исполнить.

Metamodel решает эту проблему, создавая специальные описательные классы, которые используются в Criteria API вместо имён полей. Сам Metamodel класс выглядит примерно вот так:

```
@StaticMetamodel(Company.class)
public abstract class Company_ extends AbstractIdentifiableObject_ {
       public static volatile SingularAttribute<Company, String> name;
       public static volatile CollectionAttribute<Company, Person> workers;
}
```

В Metamodel классе описываются, какие поля присутствуют в сущности, какого они типа, коллекция это или нет и т.д. Для каждой сущности создаётся свой класс Metamodel.

Создаются классы Metamodel разумеется не вручную. То есть можно их и вручную создать, но тогда пропадает автоматичность проверки и теряется смысл всей этой затеи. Обычно же классы Metamodel генерируются на этапе компиляции тем или иным методом. Конкретная реализация генерации зависит от конкретной реализации JPA и может меняться.

37. Расскажите про проблему N+1 Select и путях ее решения.

Источники:
- Vlad Mihalcea - N+1 query problem
- LogicBig - JPQL FETCH JOIN
- Baeldung - FetchMode in Hibernate
- Hibernate - Fetching

Проблема N+1 запросов возникает, когда получение данных из БД выполняется за N дополнительных SQL-запросов для извлечения тех же данных, которые могли быть получены при выполнении основного SQL-запроса.

Допустим у нас есть две сущности Post и PostComment. В БД имеется 4 Post и у каждого из них по одному PostComment:

```
@Entity(name = "Post")
@Table(name = "post")
public class Post {
    @Id
    private Long id;
    private String title;
    //Getters and setters omitted for brevity
}

@Entity(name = "PostComment")
@Table(name = "post_comment")
public class PostComment {
    @Id
    private Long id;
    @ManyToOne
    private Post post;
    private String review;
    //Getters and setters omitted for brevity
}
```

N+1 при FetchType.EAGER

Так как у @ManyToOne план извлечения по умолчанию - EAGER, то при получении из БД сущности PostComment немедленно будет загружена связанная с ней сущность Post:

```
List<PostComment> comments = entityManager.createQuery(
      """select pc from PostComment pc""", PostComment.class)
      .getResultList();
```

Этот SQL-запрос приведет к проблеме N+1 и выполнит больше запросов, чем нужно:

```
SELECT
    pc.id AS id1_1_,
    pc.post_id AS post_id3_1_,
    pc.review AS review2_1_
FROM
    post_comment pc
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=1
```

```
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=2
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=3
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=4
```

Обратите внимание на дополнительные 4 оператора SELECT, которые выполняются, потому что 4 сущности Post должны быть извлечены из БД до возврата списка из 4 сущностей PostComment с инициализированными полями Post. Автоматически Hibernate делает это не очень хорошо, порождая количество запросов в БД, равное N+1, а именно один запрос для получения PostComment, и четыре запроса для получения Post для каждого PostComment. В нашем случае это проблема 4+1.

N+1 при FetchType.LAZY

Даже если мы явно переключимся на использование FetchType.LAZY для всех ассоциаций, мы всё равно можем столкнуться с проблемой N+1. Явно укажем над полем Post план извлечения - LAZY:

```
@ManyToOne(fetch = FetchType.LAZY)
private Post post;
```

Теперь, когда мы выбираем все сущности PostComment, Hibernate выполнит одну инструкцию SQL:

```
SELECT
    pc.id AS id1_1_,
    pc.post_id AS post_id3_1_,
    pc.review AS review2_1_
FROM
    post_comment pc
```

Но, если в этом же контексте персистентности, мы обратимся к сущностям Post в PostComment:

```
for(PostComment comment : comments) {
    LOGGER.info(
        "The Post '{}' got this review '{}'",
        comment.getPost().getTitle(),
        comment.getReview()
    );
}
```

То мы опять получим проблему N+1:

```
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=1
-- The Post 'High-Performance Java Persistence - Part 1' got this review
-- 'Excellent book to understand Java Persistence'
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=2
-- The Post 'High-Performance Java Persistence - Part 2' got this review
-- 'Must-read for Java developers'
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=3
-- The Post 'High-Performance Java Persistence - Part 3' got this review
-- 'Five Stars'
SELECT p.id AS id1_0_0_, p.title AS title2_0_0_ FROM post p WHERE p.id=4
-- The Post 'High-Performance Java Persistence - Part 4' got this review
-- 'A great reference book'
```

Поскольку Post извлекается лениво, вторая группа SQL-запросов, состоящая в нашем случае из 4 штук, будет выполняться при обращении к ленивым полям Post у каждого PostComment.

Решения проблемы N+1:

1. *JOIN FETCH*

И при FetchType.EAGER и при FetchType.LAZY нам поможет JPQL-запрос с JOIN FETCH. Опцию «FETCH» можно использовать в JOIN (INNER JOIN или LEFT JOIN) для выборки связанных объектов в одном запросе вместо дополнительных запросов для каждого доступа к ленивым полям объекта.

```
List<PostComment> comments = entityManager.createQuery("""
    select pc
    from PostComment pc
    join fetch pc.post p
    """, PostComment.class)
.getResultList();

for(PostComment comment : comments) {
    LOGGER.info(
        "The Post '{}' got this review '{}'",
        comment.getPost().getTitle(),
        comment.getReview()
    );
}
```

На этот раз Hibernate выполнит одну инструкцию SQL:

```
SELECT
    pc.id as id1_1_0_,
    pc.post_id as post_id3_1_0_,
    pc.review as review2_1_0_,
    p.id as id1_0_1_,
    p.title as title2_0_1_
FROM
    post_comment pc
INNER JOIN
    post p ON pc.post_id = p.id
```

Использование LEFT JOIN FETCH аналогично JOIN FETCH, только будут загружены все сущности из таблицы PostComment, даже те, у которых нет связанной сущности Post (в нашем случае пример не логичный, но понятный).

2. *EntityGraph*

В случаях, когда нам нужно получить по-настоящему много данных, и у нас jpql запрос - лучше всего использовать EntityGraph.

3. *@Fetch(FetchMode.SUBSELECT)*

Это Аннотация Hibernate, в JPA её нет. Можно использовать только с коллекциями. Будет сделан один sql-запрос для получения корневых сущностей и, если в контексте персистентности будет обращение к ленивым полям-коллекциям, то выполнится еще один запрос для получения связанных коллекций:

```
@Entity
public class Customer {
    @Id
    @GeneratedValue
    private Long id;
    @OneToMany(mappedBy = "customer")
    @Fetch(value = FetchMode.SUBSELECT)
    private Set<Order> orders = new HashSet<>();
    // getters and setters
}
```

Первый запрос:

```
select ...
from customer customer0_
```

Второй запрос:

```
select ...
    from
        order order0_
    where
        order0_.customer_id in (
            select
                customer0_.id
            from
                customer customer0_
        )
```

4. Batch fetching

Это Аннотация Hibernate, в JPA её нет. Указывается над классом сущности или над полем коллекции с ленивой загрузкой. Будет сделан один sql-запрос для получения корневых сущностей и, если в контексте персистентности будет обращение к ленивым полям-коллекциям, то выполнится еще один запрос для получения связанных коллекций. Изменим пример:

```
@OneToMany(mappedBy = "customer")
@Fetch(value = FetchMode.SELECT)
@BatchSize(size=5)
private Set<Order> orders = new HashSet<>();
```

Например, мы знаем, что в персистентный контекст загружено 12 сущностей Customer, у которых по одному полю-коллекции orders, но так как это @OneToMany, то у них ленивая загрузка по умолчанию и они не загружены в контекст персистентности из БД. При первом обращении к какому-нибудь полю orders, нам бы хотелось, чтобы для всех 12 сущностей Customer были загружены их 12 коллекций Order, по одной для каждой. Но так как у нас @BatchSize(size=5), то Hibernate сделает 3 запроса: в первом и втором получит по пять коллекций, а в третьем получит две коллекции.

Если мы знаем примерное количество коллекций, которые будут использоваться в любом месте приложения, то можно использовать @BatchSize и указать нужное количество.

Также аннотация @BatchSize может быть указана у класса. Рассмотрим пример, где у нас есть сущность Order, у которой есть поле типа Product(не коллекция). Мы выгрузили в контекст персистентности 27 объектов Order. При обращении к полям

Product у объектов Order будет инициализировано до 10 ленивых прокси сущностей Product одновременно:

```
@Entity
class Order {
    @OneToOne(fetch = FetchType.LAZY)
    private Product product;
    ...
}

@Entity
@BatchSize(size=10)
class Product {
    ...
}
```

Хотя использовать @BatchSize лучше, чем столкнуться с проблемой запроса N+1, в большинстве случаев гораздо лучшей альтернативой является использование DTO или JOIN FETCH, поскольку они позволяют получать все необходимые данные одним запросом.

5. *HibernateSpecificMapping, SqlResultSetMapping*
   Для нативных запросов рекомендуется использовать именно их.

38. Что такое Entity Graph? Как и для чего его использовать?

Источники: Baeldung - Entity Graph
Vlad Mihalcea - Hibernate query hints

В JPA 2.1 введен EntityGraph как более сложный способ решения проблем с производительностью при загрузке данных из БД. Он позволяет определить шаблон путем группировки связанных полей, которые мы хотим получить, и позволяет нам выбирать тип графа во время выполнения.

До JPA 2.0 для загрузки связанных сущностей и коллекций мы обычно использовали FetchType.LAZY и FetchType.EAGER в качестве стратегий извлечения. Это указывало Hibernate, извлекать из БД связанные сущности и коллекции или нет. К сожалению, эта конфигурация статична и не позволяет переключаться между этими двумя стратегиями в рантайме.

Основная цель JPA Entity Graph - улучшить производительность в рантайме при загрузке базовых полей сущности и связанных сущностей и коллекций.

Вкратце, Hibernate загружает весь граф в одном SELECT-запросе, то есть все указанные связи от нужной нам сущности:

```
@Entity
public class Comment {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String reply;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn
    private Post post;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn
    private User user;
    //...
}

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;
    //...
}

@NamedEntityGraph(
  name = "post-entity-graph-with-comment-users",
  attributeNodes = {
```

```
    @NamedAttributeNode("subject"),
    @NamedAttributeNode("user"),
    @NamedAttributeNode(value = "comments", subgraph = "comments-subgraph"),
  },
  subgraphs = {
    @NamedSubgraph(
      name = "comments-subgraph",
      attributeNodes = {
        @NamedAttributeNode("user")
      }
    )
  }
)
@Entity
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String subject;
    @OneToMany(mappedBy = "post")
    private List<Comment> comments = new ArrayList<>();

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn
    private User user;

    //...
}
```

В данном примере мы создали EntityGraph и при его помощи хотим, чтобы при загрузке сущности Post загружались поля subject, user, comments. Также мы захотели, чтобы у каждой сущности comments загружалось поле user, для чего мы использовали subgraphs.

EntityGraph можно определить и без помощи аннотаций, а используя entityManager из JPA API:

```
EntityGraph<Post> entityGraph = entityManager.createEntityGraph(Post.class);
entityGraph.addAttributeNodes("subject");
entityGraph.addAttributeNodes("user");
entityGraph.addSubgraph("comments").addAttributeNodes("user");
```

JPA определяет два свойства или подсказки, с помощью которых Hibernate может выбирать стратегию извлечения графа сущностей во время выполнения:

- fetchgraph - из базы данных извлекаются только указанные в графе атрибуты. *Поскольку мы используем Hibernate, мы можем заметить, что в отличие от спецификаций JPA, атрибуты, статически настроенные как EAGER, также загружаются.*
- loadgraph - в дополнение к указанным в графе атрибутам, также извлекаются атрибуты, статически настроенные как EAGER.

В любом случае, первичный ключ и версия, если таковые имеются, всегда загружаются.

Загрузить EntityGraph можем тремя способами:

1. Используя перегруженный метод find(), который принимает Map с настройками EntityGraph:

```
EntityGraph entityGraph = entityManager.getEntityGraph("post-entity-graph");
Map<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.fetchgraph", entityGraph);
Post post = entityManager.find(Post.class, id, properties);
```

2. Используя JPQL и передав подсказку (hint):

```
EntityGraph entityGraph = entityManager.getEntityGraph("post-entity-graph-with-comment-users");
Post post = entityManager.createQuery("select p from Post p where p.id = :id",
Post.class)
  .setParameter("id", id)
  .setHint("javax.persistence.fetchgraph", entityGraph)
  .getSingleResult();
```

3. С помощью Criteria API:

```
EntityGraph entityGraph = entityManager.getEntityGraph("post-entity-graph-with-comment-users");
CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
CriteriaQuery<Post> criteriaQuery = criteriaBuilder.createQuery(Post.class);
Root<Post> root = criteriaQuery.from(Post.class);
criteriaQuery.where(criteriaBuilder.equal(root.<Long>get("id"), id));
TypedQuery<Post> typedQuery = entityManager.createQuery(criteriaQuery);
typedQuery.setHint("javax.persistence.loadgraph", entityGraph);
Post post = typedQuery.getSingleResult();
```

Все они дают следующий результат:

```
select
    post0_.id as id1_1_0_,
    post0_.subject as subject2_1_0_,
    post0_.user_id as user_id3_1_0_,
    comments1_.post_id as post_id3_0_1_,
    comments1_.id as id1_0_1_,
    comments1_.id as id1_0_2_,
    comments1_.post_id as post_id3_0_2_,
    comments1_.reply as reply2_0_2_,
    comments1_.user_id as user_id4_0_2_,
    user2_.id as id1_2_3_,
    user2_.email as email2_2_3_,
    user2_.name as name3_2_3_
from
    Post post0_
left outer join
    Comment comments1_
        on post0_.id=comments1_.post_id
left outer join
    User user2_
        on post0_.user_id=user2_.id
where
    post0_.id=?
```

В каждом из них стратегия графа (fetchgraph, loadgraph) указана как подсказка. В первом примере мы использовали Map, в то время как в двух последующих примерах мы использовали метод setHint().