1. Основные различия между JPA и JDBC (уровень абстракции)

2. Что такое хибер

3. Будет ли JPQL запрос считаться корректным HQL запросом?

4. Когда вызываешь метод createQuery() какой интерфейс получаешь

5. Основные интерфейсы JPA (ОСНОВНЫЕ 2)

6. Чем EntityManager отличается от Session

7. Разница между EntityManagerFactory и SessionFactory

8. В каком случае я могу восстановить удаленную сущность

9. Что такое PersistenceContext (это и есть кэш 1-го уровня у EntityManager)

10. Как отсоединить сущность от контекста, какими методами

11. Как можно конфигурировать хибернейт - какими способами (SessionFactory & EntityManagerFactory)

12. Основные аннотации хибера

13. @Id

14. Стратегии генерации @Id - знать все!!!!

15. Что означает стратегия @GeneratedValue.IDENTITY

16. На какой стороне инкрементится id- на стороне базы или хибернейта

17. @Column

18. @Acess

19. Что означает двусторонняя связь. Как это отобразится в коде

20. Что произойдет если не поставить mappedBy на @OneToOne @OneToMany - почему выгодно ставить mapped by на этих типах связей

21. @JoinColumn

22. @JoinTable

23. Университеты и Студенты - @MаnyToMany - можно ли обойтись просто этой аннотацией без @JoinTable и @JoinColumn

24. При аннотации @ManyToMany - ОБА ВЛАДЕЛЬЦЫ СВЯЗИ

25. Что ты знаешь про стратегии загрузки

26. Стратегии загрузки по умолчанию для всех видов связей, а также для аннотации @Basic и @Collection

27. Что такое встраиваемый класс (@Embedded)

28. Как создать составной первичный ключ - где это указывать, как это должно правильно работать

29. Для чего еще нужен @Embedded - второй случай, если опустить составной ключ

30. Как работает первый уровень кэша - когда он есть - когда нет - к чему он привязан (к какому объекту)

31. К какому объекту привязан кэш второго уровня (к EntityManagerFactory)

32. Как настроить кэш второго уровня.

33. Какой кэш еще есть. Кэш запросов - как настроить. Желательно понимать как объекты хранятся в кэше второго уровня и в кэше запросов.

34 Как контролировать объекты второго уровня кэша - как удалить как посмотреть.

35. @Basic

36. @ElementCollection

37. @OrderBy

38. @OrderColumn - как работает, где ставится

39. Различия между @OrderBy и @OrderColumn - пример с базой данных

40. OrphanRemoval

41. Почему есть Cascade Removal и orphanRemoval - есть еще один случай - когда есть разница? А при удалении разницы нет - в каком случае есть?

42. Метод unWrap()

43. Каким методом очищается кэш 1-го уровня

**Ответы на доп.вопросы**

1. Основные различия между JPA и JDBC (уровень абстракции)

Основное различие между JPA и JDBC - уровень абстракции:

JDBC - это стандарт низкого уровня для взаимодействия с базами данных посредством SQL.

JPA (Java Persistence API) это спецификация Java EE и Java SE, описывающая систему управления сохранением java объектов в таблицы реляционных баз данных в удобном виде. Сама Java не содержит реализации JPA, однако существует много реализаций данной спецификации от разных компаний (открытых и нет). Это не единственный способ сохранения java объектов в базы данных (ORM систем), но один из самых популярных в Java мире.

<https://stackoverflow.com/questions/11881548/jpa-or-jdbc-how-are-they-different>

<https://habr.com/ru/post/265061/>

2. Что такое хибер

Hibernate одна из самых популярных открытых реализаций спецификации JPA. JPA только описывает правила и API, а Hibernate реализует эти описания, впрочем у Hibernate (как и у многих других реализаций JPA) есть дополнительные возможности, не описанные в JPA (и не переносимые на другие реализации JPA).

<https://habr.com/ru/post/265061/>

Hibernate — библиотека для языка программирования Java, предназначенная для решения задач объектно-реляционного отображения (object-relational mapping — ORM). Она представляет собой свободное программное обеспечение с открытым исходным кодом (open source). Данная библиотека предоставляет легкий в использовании каркас (фреймворк) для отображения объектно-ориентированной модели данных в традиционные реляционные базы данных.

3. Будет ли JPQL запрос считаться корректным HQL запросом?

[Hibernate](https://easyjava.ru/tag/hibernate/) Query Language(JPQL) основан на Hibernate Query Language (HQL).

HQL ориентирован на запросы не к таблицам, а к классам. Любой JPQL запрос является одновременно и корректным [HQL](https://easyjava.ru/tag/hql/) запросом. Обратное может быть не верно.

<https://easyjava.ru/data/hibernate/hibernate-query-language/>

4. Когда вызываешь метод createQuery() какой интерфейс получаешь

При вызове createQuery() получаем интерфейс Query.

Это объектно-ориентированное представление запроса Hibernate. Запрос экземпляра Query получается вызовом Session.createQuery().

<https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/Query.html>

5. Основные интерфейсы JPA (ОСНОВНЫЕ 2)

На следующем рисунке показана архитектура уровня класса JPA. Он показывает основные классы и интерфейсы JPA.

1. [EntityManagerFactory](http://www.javaguides.net/2018/12/jpa-entitymanagerfactory-interface-with-example.html) - это фабричный класс [EntityManager](http://www.javaguides.net/2018/12/jpa-entitymanager-interface-with-example.html). Он создает и управляет несколькими экземплярами EntityManager.
2. [EntityManager](http://www.javaguides.net/2018/12/jpa-entitymanager-interface-with-example.html) - это интерфейс; он управляет операциями сохранения на объектах. Это работает как фабрика для экземпляра Query .
3. Entity - Сущности - это постоянные объекты, хранящиеся в виде записей в базе данных.
4. [EntityTransaction](http://www.javaguides.net/2018/12/jpa-entitytransaction-interface-with-example.html) - имеет [непосредственное](http://www.javaguides.net/2018/12/jpa-entitytransaction-interface-with-example.html) отношение к [EntityManager](http://www.javaguides.net/2018/12/jpa-entitymanager-interface-with-example.html) . Для каждого EntityManager операции поддерживаются классом [EntityTransaction](http://www.javaguides.net/2018/12/jpa-entitytransaction-interface-with-example.html) .
5. Persistence - этот класс содержит статические методы для получения EntityManagerFactory экземпляра.
6. Query - этот интерфейс реализуется каждым поставщиком JPA для получения реляционных объектов, соответствующих критериям.

<https://dzone.com/articles/introduction-to-jpa-architecture>

6. Чем EntityManager отличается от Session

Реализация JPA EntityManager является оберткой (wrap) реализации Hibernate Session. Session расширяет интерфейс EntityManager. Если Вы выбираете путь JPA, то всегда имеете возможность быстро перейти, на другие реализации JPA - EclipseLink, OpenJPA, DataNucleus. При желании, можно работая с EntityManager дотянуться через unwrap до интерфейса Session и воспользоваться его функционалом.

<https://ru.stackoverflow.com/questions/965018/hibernate-%D1%80%D0%B0%D0%B7%D0%BD%D0%B8%D1%86%D0%B0-%D0%BC%D0%B5%D0%B6%D0%B4%D1%83-session-%D0%B8-entitymanager>

<https://javarush.ru/groups/posts/1502-voprosih-na-sobesedovanie-hibernate>

7. Разница между EntityManagerFactory и SessionFactory

Hibernate SessionFactory расширяет JPA EntityManagerFactory.

Таким образом, Hibernate SessionFactory это также JPA EntityManagerFactory.

Оба SessionFactory и EntityManagerFactory содержат метаданные отображения сущностей и позволяют вам создавать Hibernate Session или EntityManager.

## Session против EntityManager

Так же как SessionFactory и EntityManagerFactory, Hibernate Session расширяет JPA EntityManager. Итак, все методы, определенные в EntityManager, доступны в Hibernate Session.

<https://javarush.ru/groups/posts/1502-voprosih-na-sobesedovanie-hibernate>

<https://stackoverflow.com/questions/5640778/hibernate-sessionfactory-vs-entitymanagerfactory>

8. В каком случае я могу восстановить удаленную сущность

Refresh

* Если статус Entity managed, то в результате операции Refresh будут восстановлены все изменения из базы данных данного Entity, также произойдет refresh всех каскадно зависимых объектов

9. Что такое PersistenceContext (это и есть кэш 1-го уровня у EntityManager)

PersistenceContext (контекст постоянства) находится между клиентским кодом и БД. Это промежуточная область, где постоянные данные преобразуются в сущности, готовые для чтения и изменения клиентским кодом.

Он отслеживает все загруженные данные, отслеживает изменения этих данных и несет ответственность за то, чтобы в конечном итоге синхронизировать любые изменения обратно в базу данных в конце транзакции.

Hibernate, использует контекст постоянства для управления жизненным циклом объекта в приложении.

Экземпляр EntityManager связан с PersistenceContext. PersistenceContext - это набор экземпляров объекта, в котором для любого постоянного идентификатора объекта существует уникальный экземпляр объекта. В PersistenceContext экземпляры сущности и их жизненный цикл управляются. EntityManager API используется для создания и удаления постоянных экземпляров сущностей, для поиска сущностей по их первичному ключу и для запросов к сущностям.

PersistenceContext - это кэш первого уровня, в котором все объекты извлекаются из базы данных или сохраняются в базе данных. PersistenceContext отслеживает любые изменения, внесенные в управляемый объект. Если что-то меняется во время транзакции, то объект помечается как грязный. Когда транзакция завершается, эти изменения сбрасываются в постоянное хранилище.

EntityManager это интерфейс, который позволяет нам взаимодействовать с контекстом постоянства. Всякий раз, когда мы используем EntityManager, мы фактически взаимодействуем с контекстом постоянства.

PersistenceContext доступны в двух типах:

* PersistenceContext в области транзакций
* PersistenceContext расширенной области

### Транзакционный контекст персистентности.

Контекст постоянства транзакции привязан к транзакции. Как только транзакция заканчивается, объекты, присутствующие в контексте постоянства, будут сброшены в постоянное хранилище.

Когда мы выполняем какую-либо операцию внутри транзакции, EntityManager проверяет PersistenceContext. Если он существует, он будет использован. В противном случае это создаст PersistenceContext.

Тип контекста персистентности по умолчанию - PersistenceContextType.TRANSACTION. Чтобы указать EntityManager использовать контекст постоянства транзакции, мы просто аннотируем его с помощью @PersistenceContext :

@PersistenceContext

private EntityManager entityManager;

Расширенный персистентный контекст

Расширенный контекст постоянства может охватывать несколько транзакций. Мы можем сохранить сущность без транзакции, но не можем сбросить ее без транзакции.

Чтобы сказать EntityManager использовать контекст персистентности расширенной области, нам нужно применить атрибут type @PersistenceContext:

@PersistenceContext(type = PersistenceContextType.EXTENDED)

private EntityManager entityManager;

<https://www.baeldung.com/jpa-hibernate-persistence-context>

10. Как отсоединить сущность от контекста, какими методами

Когда сущность только создана и записана в базу данных или когда наоборот, прочитана из базы данных, она входит в PersistenceContext и обладает неким экземпляром Session, который ей управляет. Однако из этого состояния она может внезапно перейти в состояние «отделенная» (detached). В этом состоянии сущность не связана со своим контекстом (отделена от него) и нет экземпляра Session, который бы ей управлял.

Перейти в это состояние сущность может по следующим причинам:

* Явный перевод из persisted в detached вызовом метода evict() у Session.
* Сброс контекста методом clear() у Session.
* Явное закрытие сессии методом close().
* Неявное закрытие сессии связанное с удалением объекта Session.

Над detached объектом нельзя выполнять операции, которые требуют наличия PersistenceContext.

detached сущность можно вернуть в состояние persisted вызовами merge(), lock() или update(), но не saveOrUpdate().

<https://easyjava.ru/data/hibernate/upravlenie-sushhnostyami-v-hibernate/>

11. Как можно конфигурировать хибернейт - какими способами (SessionFactory & EntityManagerFactory)

Существует четыре способа конфигурации работы с Hibernate :

— используя аннотации;

— hibernate.cfg.xml;

— hibernate.properties;

— persistence.xml.

Самый частый способ конфигурации: через аннотации и файл persistence.xml, что касается файлов hibernate.properties и hibernate.cfg.xml, то hibernate.cfg.xml главнее (если в приложение есть оба файла, то принимаются настройки из файла hibernate.cfg.xml). Конфигурация аннотациями, хоть и удобна, но не всегда возможна, например, если для разных баз данных или для разных ситуаций вы хотите иметь разные конфигурацию сущностей, то следует использовать xml файлы конфигураций.

По мимо этого хибернейт можно сконфигурировать с использованием SessionFactory или EntityManagerFactory.

При использовании JPA или Hibernate у вас есть два варианта:

1. Вы можете загрузиться с помощью встроенного механизма Hibernate и создать SessionFactory.
2. Или вы можете создать JPA EntityManagerFactory

Начальная загрузка через JPA должна быть предпочтительной. Кроме того, если вы использовали JPA, и вы вводили EntityManagerFactory через @PersistenceUnit аннотации:

@PersistenceUnit

private EntityManagerFactory entityManagerFactory;

Вы можете легко получить доступ к базовому SessionFactory используя unwrap метод:

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

То же самое можно сделать с JPA EntityManager. Если вы вводите EntityManager через @PersistenceContext аннотацию:

@PersistenceContext

private EntityManager entityManager;

Вы можете легко получить доступ к базовому, Session используя unwrap метод:

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

## Вывод

Таким образом, вам следует загружать через JPA, использовать EntityManagerFactory и EntityManager и развертывать их только в связанных с ними интерфейсах Hibernate, когда вы хотите получить доступ к некоторым специфичным для Hibernate методам, которые недоступны в JPA, например, к [извлечению объекта через его естественный идентификатор](https://vladmihalcea.com/the-best-way-to-map-a-naturalid-business-key-with-jpa-and-hibernate/).

<https://javarush.ru/groups/posts/1502-voprosih-na-sobesedovanie-hibernate>

<https://stackoverflow.com/questions/5640778/hibernate-sessionfactory-vs-entitymanagerfactory>

12. Основные аннотации хибера

Hibernate поддерживает как аннотации из JPA, так и свои собственные, которые находятся в пакете org.hibernate.annotations. Наиболее важные аннотации JPA и Hibernate:

1. @Entity: используется для указания класса как entity bean.
2. @Table: используется для определения имени таблицы из БД, которая будет отображаться на entity bean.
3. @Access: определяет тип доступа, поле или свойство. Поле — является значением по умолчанию и если нужно, чтобы hibernate использовал методы getter/setter, то их необходимо задать для нужного свойства.
4. @Id: определяет primary key в entity bean.
5. @EmbeddedId: используется для определения составного ключа в бине.
6. @Column: определяет имя колонки из таблицы в базе данных.
7. @GeneratedValue: задает стратегию создания основных ключей. Используется в сочетании с javax.persistence.GenerationType enum.
8. @OneToOne: задает связь один-к-одному между двумя сущностными бинами. Соответственно есть другие аннотации @OneToMany, @ManyToOne и @ManyToMany.
9. @Cascade: определяет каскадную связь между двумя entity бинами. Используется в связке с org.hibernate.annotations.CascadeType.
10. @PrimaryKeyJoinColumn: определяет внешний ключ для свойства. Используется вместе с org.hibernate.annotations.GenericGenerator и org.hibernate.annotations.Parameter.

13. @Id

С помощью @Id мы указываем первичный ключ (Primary Key) данного класса.

Типы переменных для @Id: примитивные и примитивные типы-оболочки, String, Date, BigDecimal, BigInteger.

<https://proselyte.net/tutorials/hibernate-tutorial/annotations/>

<https://www.baeldung.com/hibernate-identifiers>

<https://docs.oracle.com/javaee/7/api/javax/persistence/Id.html>

14. Стратегии генерации @Id - знать все!!!!

GenerationType.AUTO является типом генерации по умолчанию. Выбирает стратегию генерации на основе конкретного диалекта базы данных. Для большинства популярных баз данных он выбирает GenerationType.SEQUENCE.

GenerationType.IDENTITY является самым простым в использовании, но не самый лучший с точки зрения производительности. Он опирается на автоматически увеличивающийся столбец базы данных и позволяет базе данных генерировать новое значение при каждой операции вставки. Hibernate требует значения первичного ключа для каждого управляемого объекта и поэтому должен немедленно выполнить оператор вставки. Это предотвращает использование различных методов оптимизации, таких как пакетная обработка JDBC. (Идентити делает инсерт до персиста).

GenerationType.SEQUENCE использует последовательность базы данных для генерации уникальных значений.

Для получения следующего значения из последовательности базы данных требуются дополнительные операторы select. Но это не влияет на производительность для большинства приложений. (Секвенс делает селект, чтобы сгенерить id).

GenerationType.TABLE используется редко. Он моделирует последовательность, сохраняя и обновляя ее текущее значение в таблице базы данных, что требует использования пессимистических блокировок, которые помещают все транзакции в последовательный порядок. Это замедляет работу вашего приложения

<https://thoughts-on-java.org/jpa-generate-primary-keys/>

<https://habr.com/ru/post/416851/>

15. Что означает стратегия @GeneratedValue.IDENTITY

GenerationType.IDENTITY является самым простым в использовании, но не самый лучший с точки зрения производительности. Он опирается на автоматически увеличивающийся столбец базы данных и позволяет базе данных генерировать новое значение id при каждой операции вставки. Hibernate требует значения первичного ключа для каждого управляемого объекта и поэтому должен немедленно выполнить оператор вставки. Это предотвращает использование различных методов оптимизации, таких как пакетная обработка JDBC. (Идентити делает инсерт до персиста).

<https://thoughts-on-java.org/jpa-generate-primary-keys/>

16. На какой стороне инкрементится id- на стороне базы или хибернейта

GenerationType.IDENTITY - id инкрементится на стороне базы

GenerationType.SEQUENCE - id инкрементится на стороне хибера, использует дополнительные селекты, чтобы запросить id

<https://thoughts-on-java.org/jpa-generate-primary-keys/>

17. @Column

@Column нужна чтобы указать детали столбца в таблице.

@Column аннотации имеет много атрибутов, такие как name, length, nullable и unique.

Элемент name указывает имя столбца в таблице. Элемент length указывает его длину. атрибут nullable определяет, является ли элемент обнуляемым, и unique атрибут определяет, является ли уникальным столбец.

Если мы не укажем эту аннотацию, имя поля будет считаться именем столбца в таблице.

<https://www.baeldung.com/jpa-entities>

<https://proselyte.net/tutorials/hibernate-tutorial/annotations/>

<https://docs.oracle.com/javaee/7/api/javax/persistence/Column.html>

18. @Acess

@Access: определяет тип доступа, поле или свойство. Поле — является значением по умолчанию и если нужно, чтобы hibernate использовал методы getter/setter, то их необходимо задать для нужного свойства.

<https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html>

<https://habr.com/ru/post/265061/>

<https://thoughts-on-java.org/access-strategies-in-jpa-and-hibernate/>

19. Что означает двусторонняя связь. Как это отобразится в коде

У нас есть таблица для каждой корзины и еще одна таблица для каждого товара. В одной корзине может быть много товаров, поэтому здесь у нас есть отображение «один ко многим» .

То, как это работает на уровне базы данных, заключается в том, что у нас есть cart\_id в качестве первичного ключа в таблице корзины, а также cart\_id в качестве внешнего ключа в элементах .

И способ, которым мы делаем это в коде, с @OneToMany .

Давайте сопоставим класс Cart объекту Items таким образом, чтобы он отражал отношения в базе данных:

@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

}

Мы также можем добавить ссылку на корзину в пунктах, используя @ManyToOne, что делает это двунаправленным отношением. Двунаправленный означает, что мы можем получить доступ к предметам из корзины, а также к корзине из предметов.

Свойство mappedBy - это то, что мы используем, чтобы сообщить Hibernate, какую переменную мы используем для представления родительского класса в нашем дочернем классе.

@OneToMany используется для определения свойства в классе Items, которое будет использоваться для сопоставления переменной mappedBy. Вот почему у нас есть свойство с именем «cart» в классе Items:

@ManyToOne связана с переменной класса Cart. Аннотация @JoinColumn ссылается на сопоставленный столбец.

<https://www.baeldung.com/hibernate-one-to-many>

20. Что произойдет если не поставить mappedBy на @OneToOne @OneToMany - почему выгодно ставить mapped by на этих типах связей

mappedBy делает связь двунаправленной. Без mappedBy связь будет однонаправленной.

После того, как мы определили сторону-владельца отношений, Hibernate уже располагает всей информацией, необходимой для отображения этих отношений в нашей базе данных. Чтобы сделать эту ассоциацию двунаправленной, все, что нам нужно сделать, это определить сторону ссылки. Обратная или ссылочная сторона просто отображается на сторону-владельца.

Мы можем легко использовать mappedBy атрибут @OneToMany аннотацию , чтобы сделать это. Итак, давайте определим нашу сущность Employee :

@Entity

@Table(name="ITEMS")

public class Items {

//...

@ManyToOne

@JoinColumn(name="cart\_id", nullable=false)

private Cart cart;

public Items() {}

// getters and setters

}

Здесь значение mappedBy является именем атрибута сопоставления ассоциации на стороне владельца. Благодаря этому мы установили двустороннюю связь между нашими сотрудниками и сотрудниками электронной почты.

<https://www.baeldung.com/jpa-joincolumn-vs-mappedby>

21. @JoinColumn

Указывает столбец для присоединения к связной сущности или коллекции элементов. Если сама аннотация @JoinColumn имеет значение по умолчанию, то предполагается наличие одного столбца соединения и применяются значения по умолчанию.

<https://docs.oracle.com/javaee/7/api/javax/persistence/JoinColumn.html>

@JoinColumn - в связях вида One2Many/Many2One сторона, находящаяся в собственности обычно называется стороной “многих”. Обычно, это сторона, которая держит внешний ключ. Эта аннотация, фактически, описывает физический (как в бд) маппинг на стороне “многих”. В атрибут name этой аннотации дается название колонки, которая будет присоединена к таблице “многих” и которая будет заполняться значениями первичных ключей из таблицы-собственника. Таким образом - мы даём название для внешнего ключа. По факту - использование этой аннотации опционально, т.к. хибернейт, проанализировав сущность - поймет сам, что в таблице для этого класса нужно создать колонку, обозначающую внешний ключ, и даст ей соответствующее название “{name}\_id”.

<https://docs.google.com/document/d/1QXEv9hnVRFHaPlPMF3CEIaxL8dQrWYx4MI5PLLj14Cc/edit>

22. @JoinTable

Определяет сопоставление ассоциации. Он применяется к владельцу ассоциации.

@JoinTable обычно используется при отображении связей «многие ко многим» и однонаправленных связей «один ко многим». Он также может использоваться для сопоставления двунаправленных ассоциаций «многие к одному» или «один ко многим», однонаправленных связей «многие к одному» и связей «один к одному» (как двунаправленных, так и однонаправленных).

Когда @JoinTable используется при отображении отношения с встраиваемым классом на стороне-владельце отношения, содержащая сущность, а не встраиваемый класс считается владельцем отношения.

Если @JoinTable аннотация отсутствует, применяются значения по умолчанию для элементов аннотации.

<https://docs.oracle.com/javaee/7/api/javax/persistence/JoinTable.html>

23. Университеты и Студенты - @MаnyToMany - можно ли обойтись просто этой аннотацией без @JoinTable и @JoinColumn

Вроде как нет, т.к. нам нужна третья таблица для связи ManyToMany (связать первичный и внешний ключ).

<https://stackoverflow.com/questions/7979382/how-to-create-join-table-with-jpa-annotations>

24. При аннотации @ManyToMany - ОБА ВЛАДЕЛЬЦЫ СВЯЗИ

Настройка каскадов этой связи немного сложнее т.к. связь по умолчанию является двунаправленной, и, более того, в этой связи зачастую невозможно выделить собственность и собственника, т.к. границы между ними сильно размыт. Получается картина, когда обе стороны могут передавать каскадные изменения друг на друга.

Важно не использовать на связи @ManyToMany CascadeType.All, т.к. последний включает в себя каскад на Remove, а этот каскад заставляет удалять хибернейт данные не только из смежной таблицы, но и дальше, из таблицы сущности, которая привязана к данной. А за этим, если к этой сущности были привязаны ещё несколько других - произойдет и их удаление из базы

<https://docs.google.com/document/d/1QXEv9hnVRFHaPlPMF3CEIaxL8dQrWYx4MI5PLLj14Cc/edit>

25. Что ты знаешь про стратегии загрузки

В JPA описаны два типа fetch стратегии:

1) LAZY — данные поля будут загружены только во время первого доступа к этому полю

2) EAGER — данные поля будут загружены немедленно

<https://habr.com/ru/post/265061/>

26. Стратегии загрузки по умолчанию для всех видов связей, а также для аннотации @Basic и @Collection

EAGER для @Basic и ToOne

LAZY для @Collection и ToMany

<https://docs.jboss.org/hibernate/orm/5.1/userguide/html_single/chapters/domain/collections.html>

<https://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/collections.html>

27. Что такое встраиваемый класс (@Embedded)

Встраиваемый (Embeddable) класс это класс который не используется сам по себе, только как часть одного или нескольких Entity классов. Entity класс могут содержать как одиночные встраиваемые классы, так и коллекции таких классов. Также такие классы могут быть использованы как ключи или значения map. Во время выполнения каждый встраиваемый класс принадлежит только одному объекту Entity класса и не может быть использован для передачи данных между объектами Entity классов (то есть такой класс не является общей структурой данных для разных объектов). В целом, такой класс служит для того чтобы выносить определение общих атрибутов для нескольких Entity, можно считать что JPA просто встраивает в Entity вместо объекта такого класса те атрибуты, которые он содержит.

1. Такие классы должны удовлетворять тем же правилам что Entity классы, за исключением того что они не обязаны содержать первичный ключ и быть отмечены аннотацией Entity

2. Embeddable класс должен быть помечен аннотацией @Embeddable или описан в XML файле конфигурации JPA,

<https://www.baeldung.com/jpa-embedded-embeddable>

<https://habr.com/ru/post/265061/>

28. Как создать составной первичный ключ - где это указывать, как это должно правильно работать

Допустимые типы атрибутов, входящих в первичный ключ:

1. примитивные типы и их обертки Java

2. строки

3. BigDecimal и BigInteger

4. java.util.Date и java.sql.Date

В случае автогенерируемого первичного ключа (generated primary keys) допустимы

1. только числовые типы

В случае использования других типов данных в первичном ключе, он может работать только для некоторых баз данных, т.е. становится непереносимым (not portable)

<https://habr.com/ru/post/265061/>

@EmbeddedId указывает на поле составного первичного ключа, а @Embeddable объявляет класс составным ключом.

@Embeddable

public class BillingAddress implements Serializable {...}

@Entity

@Table(name = "PURCHASE\_ORDERS")

@IdClass(BillingAddress.class)

public class PurchaseOrder {...}

Обратите внимание, что есть некоторые ключевые требования, которым должен соответствовать класс составного ключа:

Мы должны пометить его с помощью @Embeddable.

Он должен реализовать java.io.Serializable

Мы должны обеспечить реализацию hashCode() и Equals() методы

Ни одно из полей не может быть сущностью

<https://www.baeldung.com/jpa-many-to-many>

<https://www.ibm.com/developerworks/ru/library/os-hibernatejpa/index.html>

29. Для чего еще нужен @Embedded - второй случай, если опустить составной ключ

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

<https://habr.com/ru/post/265061/>

30. Как работает первый уровень кэша - когда он есть - когда нет - к чему он привязан (к какому объекту)

Кэширование - это средство, предоставляемое средами ORM, которое помогает пользователям быстро запустить веб-приложение, а сама структура помогает сократить количество запросов к базе данных за одну транзакцию. Hibernate достигает второй цели, внедряя кэш первого уровня.

Кэш первого уровня в hibernate включен по умолчанию, и вам не нужно ничего делать, чтобы эта функция работала. На самом деле, вы не можете отключить его даже принудительно.

Кэш первого уровня легко понять, если мы понимаем тот факт, что он связан с объектом Session. Как мы знаем, объект сеанса создается по требованию из фабрики сеансов и теряется при закрытии сеанса. Аналогично, кэш первого уровня, связанный с объектом сеанса, доступен только до тех пор, пока объект сеанса не станет активным. Он доступен только для объекта сеанса и не доступен для любого другого объекта сеанса в любой другой части приложения.

## Важные факты

1. Кэш первого уровня связан с объектом Session, а другие объекты сеанса в приложении его не видят.
2. Область действия объектов кэша имеет сессию. Как только сессия закрыта, кэшированные объекты исчезают навсегда.
3. Кэш первого уровня включен по умолчанию, и вы не можете его отключить.
4. Когда мы запрашиваем объект в первый раз, он извлекается из базы данных и сохраняется в кэше первого уровня, связанном с сессией хибернейта.
5. Если мы снова запросим тот же объект с тем же объектом сеанса, он будет загружен из кэша, и никакой SQL-запрос не будет выполнен.
6. Загруженный объект можно удалить из сеанса с помощью метода evict(). Следующая загрузка этого объекта снова вызовет базу данных, если она была удалена с помощью метода evict().
7. Весь кэш сеанса можно удалить с помощью метода clear(). Это удалит все сущности, хранящиеся в кэше.

Несколько фактов про кэш первого уровня:   
 1. Кэш первого уровня не является потокобезопасным.  
 2. Кэш первого уровня привязан к сессии и уничтожается следом за уничтожением сессии.

Из этого следует один важный вывод: кэш первого уровня не является средством оптимизации большого количества повторяющихся запросов на выборку со стороны клиента, т.к. каждый запрос будет обрабатываться в отдельной транзакции, на которую будет выделен новый объект entityManager, который связан напрямую с новой сессией. Соответственно, на 20 одинаковых запросов пользователя будет создано 20 entityManager и 20 сессий. Будет выделено 20 транзакций, даже если запросы обрабатываются и поступают одновременно.

Кэш первого уровня нужен:  
 1. Для сохранения целостности данных  
 2. Оптимизации запросов на изменение/удаление  
 3. Оптимизация запросов на выборку в рамках одной транзакции

В пределах жизненного цикла одной сессии и в рамках одной транзакции мы можем изменить внутреннее состояние сущности неограниченное количество раз, каждое изменение будет вноситься в кэш первого уровня. Но в базу запрос отправится только тогда, когда будет сделан комит транзакции. В базу отправятся те данные, которые содержит сущность на момент последнего изменения. До тех пор, пока транзакция не будет закончена - все изменения будут храниться в кэше. Даже если мы вызовем 20 раз метод setField() у любой сущности - в базу в итоге отправится только один запрос.

Если же мы вынуждены читать в рамках одной транзакции несколько раз одни и те же данные, то, единожды загрузив данные запросом из базы мы будем в дальнейшем работать с данными внутри кэша, не повторяя дополнительных запросов. Например, если достать List<User> и затем достать конкретного юзера с id=2, то запрос в базу не будет произведен, т.к. список всех пользователей уже лежит в кэше. Так же, если мы, уже после того как достали пользователя с id=2 изменили 10 раз его имя, а затем снова выберем список всех пользователей - мы и в этом случае не получим дополнительных запросов. В описанном выше случае будет произведено только два запроса: на выборку списка всех пользователей в самом начала и один запрос на изменение состояния пользователя уже в конце транзакции.

<https://howtodoinjava.com/hibernate/understanding-hibernate-first-level-cache-with-example/>

31. К какому объекту привязан кэш второго уровня (к EntityManagerFactory)

Кэш второго уровня создается в области фабрики EntityManagerFactory и доступен для использования во всех EntityManager, которые создаются с использованием этой конкретной фабрики.

Это также означает, что после закрытия фабрики весь кэш, связанный с ним, умирает, а менеджер кэша также закрывается.

Кроме того, это также означает, что если у вас есть два экземпляра фабрики, в вашем приложении будет два менеджера кэша, и при доступе к кэшу, хранящемуся в физическом хранилище, вы можете получить непредсказуемые результаты, такие как пропадание кеша.

1. Всякий раз, когда сессия пытается загрузить объект, самое первое место, где он ищет кэшированную копию объекта в кэше первого уровня.
2. Если кэшированная копия объекта присутствует в кэше первого уровня, она возвращается как результат метода загрузки.
3. Если в кэше первого уровня нет кэшированной сущности, то для кэшированной сущности ищется кэш второго уровня.
4. Если кэш второго уровня имеет кэшированный объект, он возвращается как результат метода load(). Но перед возвратом объекта он также сохраняется в кэше первого уровня, так что при следующем вызове метода загрузки объект будет возвращен из самого кэша первого уровня, и больше не потребуется обращаться в кэш второго уровня.
5. Если объект не найден в кэше первого уровня и кэше второго уровня, то выполняется запрос к базе данных, и объект сохраняется на обоих уровнях кэша перед возвратом в качестве ответа метода load().
6. Кэш второго уровня проверяет себя для измененных объектов.
7. Если какой-либо пользователь или процесс вносят изменения непосредственно в базу данных, то само по себе кэширование второго уровня не может обновляться до тех пор, пока не истечет время «timeToLiveSeconds» для этой области кэша. В этом случае хорошей идеей будет сделать недействительным весь кеш и позволить hibernate снова построить кэш.

<https://howtodoinjava.com/hibernate/how-hibernate-second-level-cache-works/>

32. Как настроить кэш второго уровня.

Со следующими двумя свойствами мы сообщаем Hibernate, что кэширование L2 включено, и даем ему имя класса фабрики региона:

hibernate.cache.use\_second\_level\_cache=true

hibernate.cache.region.factory\_class=org.hibernate.cache.ehcache.EhCacheRegionFactory

Чтобы сделать объект пригодным для кэширования второго уровня, мы помечаем его аннотацией @Cache или @Cacheable, специфичной для Hibernate, и указываем [стратегию параллельного](https://www.baeldung.com/hibernate-second-level-cache#cacheConcurrencyStrategy) использования [кэша](https://www.baeldung.com/hibernate-second-level-cache#cacheConcurrencyStrategy):

1) ALL — все Entity могут кэшироваться в кэше второго уровня

2) NONE — кеширование отключено для всех Entity

3) ENABLE\_SELECTIVE — кэширование работает только для тех Entity, у которых установлена аннотация Cacheable(true), для всех остальных кэширование отключено

4) DISABLE\_SELECTIVE — кэширование работает для всех Entity, за исключением тех у которых установлена аннотация Cacheable(false)

5) UNSPECIFIED — кеширование не определенно, каждый провайдер JPA использует свою значение по умолчанию для кэширования

<https://www.baeldung.com/hibernate-second-level-cache>

33. Какой кэш еще есть. Кэш запросов - как настроить. Желательно понимать как объекты хранятся в кэше второго уровня и в кэше запросов.

Hibernate также поддерживает QueryCache, который может хранить результаты запроса. Вам необходимо активировать его в файле persistence.xml, установив для параметра

hibernate.cache.use\_query\_cache=true

и определив

hibernate.cache.region.factory\_class

Кроме того, вам также необходимо активировать кэширование для конкретного запроса, для которого вы хотите кэшировать результаты, вызывая

setCacheable(true)

<https://thoughts-on-java.org/hibernate-tips-use-querycache-avoid-additional-queries/>

34 Как контролировать объекты второго уровня кэша - как удалить как посмотреть.

Сохранение или обновление элемента:

save()

update()

saveOrUpdate()

Получение предмета:

load()

get()

list()

iterate()

scroll()

Состояние объекта синхронизируется с базой данных при вызове метода flush(). Чтобы избежать этой синхронизации, вы можете удалить объект и все коллекции из кэша первого уровня с помощью evict() метода. Чтобы удалить все элементы из кэша сеанса, используйте метод Session.clear():

ScrollableResult cats = sess.createQuery("from Cat as cat").scroll(); //a huge result set

while ( cats.next() ) {

Cat cat = (Cat) cats.get(0);

doSomethingWithACat(cat);

sess.evict(cat);

}

Определение того, принадлежит ли элемент кешу сеанса. Сеанс предоставляет contains() метод для определения того, принадлежит ли экземпляр кешу сеанса.

<https://docs.jboss.org/hibernate/orm/4.3/devguide/en-US/html/ch06.html>

35. @Basic

## @Basic — указывает на простейший тип маппинга данных на колонку таблицы базы данных. Также в параметрах аннотации можно указать fetch стратегию доступа к полю и является ли это поле null или нет.

## <https://habr.com/ru/post/265061/>

Самый простой тип сопоставления со столбцом базы данных. Базовая аннотация может быть применена к постоянному свойству или переменной экземпляра любого из следующих типов: примитивные типы Java, оболочки примитивных типов, String, BigInteger, BigDecimal, Date, Calendar, Time, Timestamp, byte[], Byte[], char[], Character[], enums, и любой другой тип, который реализует java.io.Serializable.

Использование базовой аннотации является необязательным для постоянных полей и свойств этих типов. Если базовая аннотация не указана для такого поля или свойства, то будут применяться значения по умолчанию базовой аннотации.

<https://docs.oracle.com/javaee/7/api/javax/persistence/Basic.html>

36. @ElementCollection

Определяет коллекцию экземпляров базового типа или встраиваемого класса. Должен быть указан, если коллекция должна отображаться с помощью таблицы коллекции.

<https://docs.jboss.org/hibernate/jpa/2.1/api/javax/persistence/ElementCollection.html>

37. @OrderBy

Определяет порядок элементов коллекции, оцениваемой ассоциацией или коллекцией элементов, в тот момент, когда ассоциация или коллекция извлекаются.

<https://docs.jboss.org/hibernate/jpa/2.1/api/javax/persistence/OrderBy.html>

@OrderBy могут быть применены к элементу коллекции. Когда OrderBy применяется к коллекции элементов базового типа, порядок будет по значению базовых объектов, а имя свойства или поля не используется. При указании порядка для коллекции элементов встраиваемого типа необходимо использовать точечную нотацию для указания атрибута или атрибутов, которые определяют порядок.

<https://docs.jboss.org/hibernate/jpa/2.1/api/javax/persistence/OrderBy.html>

38. @OrderColumn - как работает, где ставится

Указывает столбец, который используется для поддержания постоянного порядка списка. @OrderColumn указывается в отношении OneToMany или ManyToMany или в коллекции элементов. @OrderColumn указывается на стороне отношения, ссылающейся на коллекцию, которая должна быть упорядочена. <https://docs.jboss.org/hibernate/jpa/2.1/api/javax/persistence/OrderColumn.html>

39. Различия между @OrderBy и @OrderColumn - пример с базой данных

@OrderBy в запросе отсортирует, а в кэше вернет неотсортированный порядок. @OrderedColumn сортирует данные с учетом данных в колонке, и в кеше и в запросе.

Указанный порядок @OrderBy применяется только во время выполнения при получении результата запроса.

@OrderColumn приводит к постоянному упорядочению соответствующих данных.

<https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/order-by-annotation.html>

40. OrphanRemoval

@OrphanRemoval - управляет поведением осиротевшими сущностями.

OrphanRemoval: если мы вызовем setOrders(null), Order энтити будет удалена из БД автоматически.

Если каскад ремув, то setOrders(null), Order НЕ БУДЕТ УДАЛЕНА из БД автоматически.

Удаление сирот в отношениях

Когда целевой объект в отношении «один-к-одному» или «один-ко-многим» удаляется из отношения, часто желательно каскадно удалить операцию для целевого объекта. Такие целевые объекты считаются «сиротами», а атрибут orphanRemoval может использоваться для указания того, что потерянные объекты должны быть удалены. Например, если в заказе много позиций, и одна из них удалена из заказа, удаленная позиция считается сиротой. Если для orphanRemoval установлено значение true, объект позиции будет удален при удалении позиции из заказа.

<https://docs.oracle.com/cd/E19798-01/821-1841/giqxy/>

41. Почему есть Cascade Removal и orphanRemoval - есть еще один случай - когда есть разница? А при удалении разницы нет - в каком случае есть?

CascadeType определяет каскадные операции, которые применяются в элементе каскада аннотаций отношений.

Пример : позиция является частью заказа; если заказ удален, позиция также должна быть удалена. Это называется каскадным отношением удаления.

orphanRemoval - когда целевой объект в отношении один-к-одному или один-ко-многим удаляется из отношения.

Пример: если в заказе много позиций, и одна из них удалена из заказа, удаленная позиция считается сиротой. Если для orphanRemoval установлено значение true, объект позиции будет удален при удалении позиции из заказа.

Когда использовать:

Каскадное удаление удаляет все дочерние элементы при удалении родителя.

Таким образом, если вы удалите пользовательскую сущность, JPA удалит также все его фотографии.

<https://agritsik.wordpress.com/2015/08/06/orphan-removal-vs-cascade-delete-or-how-to-delete-related-entities/>

42. Метод unWrap()

JPA обеспечивает легкий доступ к API базовых реализаций. EntityManager и EntityManagerFactory обеспечивают разворачивают метод, который возвращает соответствующие классы реализации JPA. В случае Hibernate это Session и SessionFactory.

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

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

В первой строке я получаю текущий сеанс Hibernate от EntityManager . Поэтому я называю UnWrap метод на EntityManager и обеспечить сеанс класса в качестве параметра.

Вторая строка выглядит очень похоже. Я получаю EntityManagerFactory для текущего EntityManager и вызываю метод unwrap специфичный для Hibernate класс SessionFactory.

Эти классы предоставляют вам полный доступ к проприетарным функциям Hibernate, таким как поддержка [Streams](https://thoughts-on-java.org/get-query-results-stream-hibernate-5/) и [Optional](https://thoughts-on-java.org/use-java-8-optional-hibernate/).

<https://thoughts-on-java.org/hibernate-tips-access-hibernate-apis-jpa/>

43. Каким методом очищается кэш 1-го уровня

Это делается с помощью двух методов:

evict()

clear()

Здесь evict() используется для удаления конкретного объекта из кэша, связанного с сеансом, а метод clear() используется для удаления всех кэшированных объектов, связанных с сеансом.

<https://howtodoinjava.com/hibernate/understanding-hibernate-first-level-cache-with-example/>