Оглавление

Hibernate

1. Что такое ORM? Что такое JPA? Что такое Hibernate?	**ORM**(Object Relational Mapping) - это концепция преобразования данных из объектно-ориентированного языка в реляционные БД и наоборот. **JPA**(Java Persistence API) - это стандартная для Java спецификация, описывающая принципы ORM. JPA не умеет работать с объектами, а только определяет правила как должен действовать каждый провайдер (Hibernate, EclipseLink), реализующий стандарт JPA **Hibernate** - библиотека, являющаяся реализацией этой спецификации, в которой можно использовать стандартные API-интерфейсы JPA.
	Важные интерфейсы Hibernate: **Session** - обеспечивает физическое соединение между приложением и БД. Основная функция - предлагать DML-операции для экземпляров сущностей. **SessionFactory** - это фабрика для объектов Session. Обычно создается во время запуска приложения и сохраняется для последующего использования. Является потокобезопасным объектом и используется всеми потоками приложения. **Transaction** - однопоточный короткоживущий объект, используемый для атомарных операций. Это абстракция приложения от основных JDBC транзакций. Session может занимать несколько Transaction в определенных случаях, является необязательным API. **Query** - интерфейс позволяет выполнять запросы к БД. Запросы написаны на HQL или на SQL.
2. Что такое EntityManager?	EntityManager интерфейс JPA, который описывает API для всех основных операций над Entity, а также для получения данных и других сущностей JPA. Основные операции: 1) Операции над Entity: persist (добавление Entity), merge (обновление), remove (удаления), refresh (обновление данных), detach (удаление из управление JPA), 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, clear, isOpen, getProperties, setProperty.
	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?	**Entity** это легковесный хранимый объект бизнес логики. Основная программная сущность это entity-класс, который так же может использовать дополнительные классы, которые могут использоваться как вспомогательные классы или для сохранения состояния entity.
	1) Entity класс должен быть помечен аннотацией Entity или описан в XML файле 2) Entity класс должен содержать public или protected конструктор без аргументов (он также может иметь конструкторы с аргументами) - при получении данных из БД и формировании из них объекта сущности, Hibernate должен создать этот самый объект сущности, 3) Entity класс должен быть классом верхнего уровня (top-level class), 4) Entity класс не может быть enum или интерфейсом, 5) Entity класс не может быть финальным классом (final class), 6) Entity класс не может содержать финальные поля или методы, если они участвуют в маппинге (persistent final methods or persistent final instance variables), 7) Если объект Entity класса будет передаваться по значению как отдельный объект (detached object), например через удаленный интерфейс (through a remote interface), он так же должен реализовывать Serializable интерфейс 8) Поля Entity класс должны быть напрямую доступны только методам самого Entity класса и не должны быть напрямую доступны другим классам, использующим этот Entity. Такие классы должны обращаться только к методам (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

4. Может ли абстрактный класс быть Entity?	Может, при этом он сохраняет все свойства Entity, за исключением того что его нельзя непосредственно инициализировать.
5. Может ли Entity класс наследоваться от не Entity классов (non-entity classes)?	Может
6. Может ли Entity класс наследоваться от других Entity классов?	Может
7. Может ли не Entity класс наследоваться от Entity класса?	Может
8. Что такое встраиваемый (Embeddable) класс? Какие требования JPA устанавливает к встраиваемым (Embeddable) классам?	**Embeddable класс - это класс, который не используется сам по себе, а является частью одного или нескольких Entity-классов.** Entity-класс может 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.
	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. А поле этого класса в Entity аннотацией @Embedded
	Embeddable-класс может содержать другой встраиваемый класс.

	Встраиваемый класс может содержать связи с другими Entity или коллекциями Entity, если такой класс не используется как первичный ключ или ключ map'ы.
9. Что такое Mapped Superclass?	Mapped Superclass - это класс, от которого наследуются Entity, он может содержать анотации JPA, однако сам такой класс не является Entity, ему не обязательно выполнять все требования установленные для Entity (например, он может не содержать первичного ключа). Такой класс не может использоваться в операциях EntityManager или Query. Такой класс должен быть отмечен аннотацией MappedSuperclass или описан в xml файле. Создание такого класса-предка оправдано тем, что мы заранее определяем ряд свойств и методов, которые должны быть определены в сущностях. Использование такого подхода позволило сократить количество кода
10. Какие три типа стратегий наследования мапинга (Inheritance Mapping Strategies) описаны в JPA?	одна таблица на всю иерархию наследования (a single table per class hierarchy) — все entity, со всеми наследниками записываются в одну таблицу, для идентификации типа entity определяется специальная колонка “discriminator column”. Например, если есть entity Animals с классами-потомками Cats и Dogs, при такой стратегии все entity записываются в таблицу Animals, но при это имеют дополнительную колонку animalType в которую соответственно пишется значение «cat» или «dog».Минусом является то что в общей таблице, будут созданы все поля уникальные для каждого из классов-потомков, которые будет пусты для всех других классов-потомков. Например, в таблице animals окажется и скорость лазанья по дереву от cats и может ли пес приносить тапки от dogs, которые будут всегда иметь null для dog и cat соответственно.
	2) Стратегия «соединения» (JOINED) — в этой стратегии каждый класс enity сохраняет данные в свою таблицу, но только уникальные поля (не унаследованные от классов-предков) и первичный ключ, а все унаследованные колонки записываются в таблицы класса-предка, дополнительно устанавливается связь (relationships) между этими таблицами, например в случае классов Animals (см.выше), будут три таблицы animals, cats, dogs, причем в cats будет записана только ключ и скорость лазанья, в dogs — ключ и умеет ли пес приносить палку, а в animals все остальные данные cats и dogs с ссылкой на соответствующие таблицы. **Минусом тут являются потери производительности от объединения таблиц (join) для любых операций.**
	3) Таблица для каждого класса (TABLE_PER_CLASS) — каждый отдельный класс-наследник имеет свою таблицу, т.е. для cats и dogs (см.выше) все данные будут записываться просто в таблицы cats и dogs как если бы они вообще не имели общего суперкласса. **Минусом является плохая поддержка полиморфизма (polymorphic relationships) и то что для выборки всех классов иерархии потребуются большое количество отдельных sql запросов или использование UNION запроса.** Для задания стратегии наследования используется аннотация Inheritance (или соответствующие блоки)
11. Как мапятся Enum'ы?	**@Enumerated(EnumType.STRING)** - означает, что в базе будут храниться имена Enum. **@Enumerated(EnumType.ORDINAL)** - в базе будут храниться порядковые номера Enum.
	Другой вариант - мы можем смапить наши enum в БД и обратно в методах с аннотациями **@PostLoad и @PrePersist. @EntityListener над классом Entity, в которой указать класс, в котором создать два метода, помеченнх этими аннотациями.** Идея 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.

	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 и поле в сущности аннотацией @Convert**.
12. Как мапятся даты (до java 8 и после)?	Аннотация **@Temporal** до Java 8, **в которой надо было указать какой тип даты мы хотим использовать.** В Java 8 и далее аннотацию ставить не нужно.
13. Как "смапить" коллекцию примитивов?	**@ElementCollection** @OrderBy 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). Если у нашей сущности коллекция не других сущностей, а базовых или встраиваемых (embeddable) типов для этих случаев в JPA имеется специальная аннотация **@ElementCollection, которая указывается в классе сущности над полем коллекции. Все записи коллекции хранятся в отдельной таблице, то есть в итоге получаем две таблицы: одну для сущности, вторую для коллекции элементов.** **При добавлении новой строки в коллекцию, она полностью очищается и заполняется заново, так как у элементов нет id. Можно решить с помощью @OrderColumn** @CollectionTable - позволяет редактировать таблицу с коллекцией, прочитать
14. Какие есть виды связей?	Существуют 4 типа связей: 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.
	Каждую из которых можно разделить ещё на два вида: 1. Bidirectional с использованием mappedBy на стороне, где указывается @OneToMany 2. Unidirectional
	Bidirectional — ссылка на связь устанавливается у всех Entity, то есть в случае OneToOne A-B в Entity A есть ссылка на Entity B, в Entity B есть ссылка на Entity A. Entity A считается владельцем этой связи (это важно для случаев каскадного удаления данных, тогда при удалении A также будет удалено B, но не наоборот). **Undirectional**- ссылка на связь устанавливается только с одной стороны, то есть в случае OneToOne A-B только у Entity A будет ссылка на Entity B, у Entity B ссылки на A не будет.

15. Что такое владелец связи?	В отношениях между двумя сущностями всегда есть одна владеющая сторона, а владеемой может и не быть, если это однонаправленные отношения. 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.
16. Что такое каскады?	Каскадирование - это когда мы выполняем какое-то действие с целевой Entity, то же самое действие будет применено к связанной Entity. JPA CascadeType: ALL - гарантируют, что все персистентные события, которые происходят на родительском объекте, будут переданы дочернему объекту. PERSIST - означает, что операции save () или persist () каскадно передаются связанным объектам. MERGE - означает, что связанные entity объединяются, когда объединяется entity-владелец.
	REMOVE - удаляет все entity, связанные с удаляемой entity. DETACH - отключает все связанные entity, если происходит «ручное отключение». REFRESH - повторно считывают значение данного экземпляра и связанных сущностей из базы данных при вызове refresh().

17. Разница между PERSIST и MERGE?	persist(entity) следует использовать с совершенно новыми объектами, чтобы добавить их в БД (если объект уже существует в БД, будет выброшено исключение EntityExistsException). Но в случае merge(entity) сущность, которая уже управляется в контексте персистентности, будет заменена новой сущностью (обновленной), и копия этой обновленной сущности вернется обратно. Но рекомендуется использовать для уже сохраненных сущностей.
18. Какие два типа fetch стратегии в JPA вы знаете?	1) **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. **В объект загружается прокси lazy-поля.** 2) **EAGER** — данные поля будут загруженны немедленно. Это поведение по умолчанию для полей, аннотированных @Basic, @ManyToOne и @OneToOne.
19. Какие четыре статуса жизненного цикла Entity объекта (Entity Instance's Life Cycle) вы можете перечислить?	**Transient (New) —** свежесозданная оператором new() сущность не имеет связи с базой данных, не имеет данных в базе данных и не имеет сгенерированных первичных ключей. **managed** - объект создан, сохранён в бд, имеет primary key, управляется JPA **detached** - объект создан, не управляется JPA. В этом состоянии сущность не связана со своим контекстом (отделена от него) и нет экземпляра Session, который бы ей управлял. **removed** - объект создан, управляется JPA, будет удален при commit-е и статус станет опять detached
20. Как влияет операция persist на Entity объекты каждого из четырех статусов?	**new** → managed, и объект будет сохранен в базу при commit-е транзакции или в результате flush операций **managed** → операция игнорируется, однако зависимые Entity могут поменять статус на managed, если у них есть аннотации каскадных изменений **detached** → exception сразу или на этапе commit-а транзакции **removed** → managed, но только в рамках одной транзакции.
21. Как влияет операция remove на Entity объекты каждого из четырех статусов?	**new** → операция игнорируется, однако зависимые Entity могут поменять статус на removed, если у них есть аннотации каскадных изменений и они имели статус managed **managed** → removed и запись объект в базе данных будет удалена при commit-е транзакции (также произойдут операции remove для всех каскадно зависимых объектов) **detached** → exception сразу или на этапе commit-а транзакции **removed** → операция игнорируется
22. Как влияет	**new** → будет создан новый managed entity, в который будут скопированы данные прошлого объекта **managed** → операция игнорируется, однако операция merge сработает на каскадно зависимые Entity, если их статус не managed

операция merge на Entity объекты каждого из четырех статусов?	**detached** → либо данные будут скопированы в существующий managed entity с тем же первичным ключом, либо создан новый managed в который скопируются данные **removed** → exception сразу или на этапе commit-а транзакции
23. Как влияет операция refresh на Entity объекты каждого из четырех статусов?	**managed** → будут восстановлены все изменения из базы данных данного Entity, также произойдет refresh всех каскадно зависимых объектов **new**, **removed**, **detached** → exception
24. Как влияет операция detach на Entity объекты каждого из четырех статусов?	**managed**, **removed** → detached. **new**, **detached** → операция игнорируется
25. Для чего нужна аннотация 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 обязательным или нет. 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. Вообще, аннотацию @Basic можно не ставить, как это и происходит по умолчанию.
	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
26. Для чего нужна аннотация Column?	@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. Коротко, в Column (колум) мы задаем constraints (констрейнтс), а в Basic (бейсик) - ФЕТЧ ТАЙП
27. Для чего нужна аннотация Access?	***Она определяет тип доступа (access type) для класса entity, суперкласса, embeddable или отдельных атрибутов, то есть как JPA будет обращаться к атрибутам entity, как к полям класса (FIELD) или как к свойствам класса (PROPERTY), имеющие гетеры (getter) и сетеры (setter).*** Определяет тип доступа к полям сущности. Для чтения и записи этих полей есть два подхода: 1. Field access (доступ по полям). При таком способе аннотации маппинга (Id, Column,...) 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.
	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. Поля, унаследованные от суперкласса, имеют тип доступа этого суперкласса. Когда у одной сущности определены разные типы доступа, то нужно использовать аннотацию @Transient для избежания дублирования маппинга.

28. Для чего нужна аннотация @Cacheable?	@Cacheable - необязательная аннотация JPA, используется для указания того, должна ли сущность храниться в кэше второго уровня. JPA говорит о пяти значениях shared-cache-mode из persistence.xml, который определяет как будет использоваться second-level cache: ❖ **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 @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.
29. Для чего нужна аннотация @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. transactional read-write nonstrict-read-write read-only
30. @Embeddable	@Embeddable - аннотация JPA, размещается над классом для указания того, что класс является встраиваемым в другие классы. @Embedded - аннотация JPA, используется для размещения над полем в классе-сущности для указания того, что мы внедряем встраиваемый класс.
31. Как смапить составной ключ?	Составной первичный ключ, также называемый составным ключом, представляет собой комбинацию из двух или более столбцов для формирования первичного ключа таблицы. @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, ComplexKey с этими полями. Затем нам нужно аннотировать сущность Account аннотацией @IdClass(ComplexKey.class). Мы также должны объявить поля из класса ComplexKey в сущности Account с такими же именами и аннотировать их с помощью @Id. public class ComplexKey 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()
	}
	
	@Entity
	@IdClass(ComplexKey.class)
	public class Account {
	@Id
	private String accountNumber;
	@Id
	private String accountType;
	// other fields, getters and setters
	}

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

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

32. Для чего нужна аннотация ID? Какие @GeneratedValue вы знаете?	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.
	Если мы хотим, чтобы значение первичного ключа генерировалось для нас автоматически, мы можем добавить первичному ключу, отмеченному аннотацией @Id, аннотацию **@GeneratedValue**. Возможно 4 варианта: **AUTO(default) -** Указывает, что Hibernate должен выбрать подходящую стратегию для конкретной базы данных, учитывая её диалект, так как у разных БД разные способы по умолчанию. Поведение по умолчанию - исходить из

	типа поля идентификатора. **IDENTITY -** для генерации значения первичного ключа будет использоваться столбец IDENTITY, имеющийся в базе данных. Значения в столбце автоматически увеличиваются вне текущей выполняемой транзакции(на стороне базы, так что этого столбца мы не увидим), что позволяет базе данных генерировать новое значение при каждой операции вставки. В промежутках транзакций сущность будет сохранена.
	SEQUENCE - тип генерации, рекомендуемый документацией Hibernate. Для получения значений первичного ключа Hibernate должен использовать имеющиеся в базе данных механизмы генерации последовательных значений (Sequence). В бд можно будет увидеть дополнительную таблицу. Но если наша БД не поддерживает тип SEQUENCE, то Hibernate автоматически переключится на тип TABLE. В промежутках транзакций сущность не будет сохранена, так как хибер возьмет из таблицы id hibernate-sequence и вернётся обратно в приложение. **SEQUENCE** - это объект базы данных, который генерирует инкрементные целые числа при каждом последующем запросе. **TABLE** - Hibernate должен получать первичные ключи для сущностей из создаваемой для этих целей таблицы, способной содержать именованные сегменты значений для любого количества сущностей. Требует использования пессимистических блокировок, которые помещают все транзакции в последовательный порядок и замедляет работу приложения.
33. Расскажите про аннотации @JoinColumn и @JoinTable? Где и для чего они используются?	@**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 @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 в таблице сущности-владельца связи, который будет содержать внешний ключ, ссылающийся на первичный ключ владеемой сущности.
	❖ @OneToMany/@ManyToOne: если не указать на владеемой стороне связи атрибут mappedBy, создается joinTable с ключами обеих таблиц. Но при этом же у владельца создается столбец с внешними ключами.
	@**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.
	@**JoinTable** используется для указания связывающей (сводной, третьей) таблицы между двумя другими таблицами.
34. Для чего нужны аннотации @OrderBy и @OrderColumn, чем они отличаются?	**@OrderBy указывает порядок, в соответствии с которым должны располагаться элементы коллекций** сущностей, базовых или встраиваемых типов **при их извлечении из БД. Если в кэше есть нужные данные, то сортировки не будет.** Так как @OrderBy просто добавляет к sql-запросу Order By, а при получении данных из кэша, обращения к бд нет. 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.
	Если это коллекция встраиваемых типов (@Embeddable), то используя точку (".") мы можем сослаться на атрибут внутри встроенного атрибута. Если это коллекция сущностей, то у аннотации @OrderBy можно указать имя

	поля сущности, по которому сортировать эти самые сущности: 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 создает в таблице столбец с индексами порядка элементов, 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.
	@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.
35. Для чего нужна аннотация Transient?	**@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 static и не final поля. Transient fields (временные поля): ❖ static и final поля сущностей; ❖ иные поля, объявленные явно с использованием Java-модификатора transient, либо JPA-аннотации @Transient.
36. Какие шесть видов блокировок (lock) описаны в спецификации JPA (или какие есть значения у enum LockModeType в JPA)?у	В порядке от самого ненадежного и быстрого, до самого надежного и медленного: 1. **NONE** — без блокировки. 2. **OPTIMISTIC** (синоним READ в JPA 1) — оптимистическая блокировка, которая работает, как описано ниже: если при завершении транзакции кто-то извне изменит поле @Version, то будет сделан RollBack транзакции и будет выброшено OptimisticLockException. 3. **OPTIMISTIC_FORCE_INCREMENT** (синоним WRITE в JPA 1) — работает по тому же алгоритму, что и LockModeType.OPTIMISTIC за тем исключением, что после commit значение поле Version принудительно увеличивается на 1. В итоге окончательно после каждого коммита поле увеличится на 2(увеличение, которое можно увидеть в Post-Update + принудительное увеличение). 4. **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. 5. **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. 6. **PESSIMISTIC_FORCE_INCREMENT** — ведёт себя как PESSIMISTIC_WRITE, но в конце транзакции увеличивает значение поля @Version, даже если фактически сущность не изменилась.
	Оптимистичное блокирование - подход предполагает, что параллельно выполняющиеся транзакции редко обращаются к одним и тем же данным и позволяет им 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 OptimisticLockException. Оптимистичное блокирование в JPA реализовано путём внедрения в сущность специального поля версии: @Version private long version; Поле, аннотирование @Version, может быть целочисленным или временнЫм. При завершении транзакции, если сущность была заблокирована оптимистично, 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. Позволяет отказатьсь от блокировок на уровне БД и делать всё с JPA.
	Пессимистичное блокирование - подход напротив, ориентирован на транзакции, которые часто конкурируют за одни и те же данные и поэтому блокирует доступ к данным 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.
	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);
37. Какие два вида кэшей (cache) вы знаете в JPA и для чего они нужны?	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.
	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.
	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 PersistenceContext 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.

	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.
38. Как работать с кешем 2 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.
	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** ❖ **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** 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.
	Стратегии 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. ❖ **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, словно они работали с ними последовательно одна транзакция за другой. Плата за это — блокировки и потеря производительности.
39. Что такое JPQL/HQL и чем он отличается от SQL?	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 - 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, то есть каждый запрос к Entity возвращает не только объекты этого Entity, но также объекты всех его классов-потомков, независимо от стратегии наследования. В JPA запрос представлен в виде javax.persistence.Query или javax.persistence.TypedQuery, полученных из EntityManager.
	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.
40. Что такое Criteria API и для чего он используется?	Начиная с версии 5.2 Hibernate Criteria API объявлен deprecated. Вместо него рекомендуется использовать JPA Criteria API. JPA Criteria API - это актуальный API, используемый только для выборки(select) сущностей из БД в более объектно-ориентированном стиле. Основные преимущества JPA Criteria API: ❖ ошибки могут быть обнаружены во время компиляции; ❖ позволяет динамически формировать запросы на этапе выполнения приложения. Основные недостатки: ❖ нет контроля над запросом, сложно отловить ошибку ❖ влияет на производительность, множество классов Для динамических запросов - фрагменты кода создаются во время выполнения - JPA Criteria API является предпочтительней.
	Вот некоторые области применения Criteria API: Criteria API поддерживает проекцию, которую мы можем использовать для агрегатных функций вроде sum(), min(), max() и т.д. Criteria API может использовать ProjectionList для извлечения данных только из выбранных колонок. Criteria API может быть использована для join запросов с помощью соединения нескольких таблиц, используя методы createAlias(), setFetchMode() и setProjection(). Criteria API поддерживает выборку результатов согласно условиям (ограничениям). Для этого используется метод add() с помощью которого добавляются ограничения (Restrictions). Criteria API позволяет добавлять порядок (сортировку) к результату с помощью метода addOrder().
41. Расскажите про проблему N+1 Select и	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.
	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

путях ее решения.	дополнительных запросов для каждого доступа к ленивым полям объекта. Лучший вариант решения для простых запросов (1-3 уровня вложенности связанных объектов). select pc from PostComment pc join fetch pc.post p
	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)** Аннотация Hibernate. **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: @Fetch(value = FetchMode.SUBSELECT) private Set<Order> orders = new HashSet<>();
	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. Количество загружаемых сущностей указывается в аннотации. @BatchSize(size=5) private Set<Order> orders = new HashSet<>();
	5. **HibernateSpecificMapping, SqlResultSetMapping** Для нативных запросов рекомендуется использовать именно их.