60 вопросов и ответов на собеседование по Hibernate

Read & Write Code

1). Что такое Hibernate Framework?

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.
Основной особенностью фреймворка (и самой полезной его частью) является то, что он представляет базу данных в форме **объекта**. Эти объекты могут быть написаны без особых знаний в SQL. Это отличная возможность, так как помогает разработчикам сэкономить много времени – это очень важно в сфере современного программирования.
Hibernate также имеет встроенное объектное отображение – это минимизирует число строк кода, необходимых для работы приложения.

2). Что такое ORM?

ORM является аббревиатурой для “**Object-related Mapping**” или “**Объектно-реляционного Отображения**”.
Это фундаментальная концепция платформы Hibernate, которая сопоставляет таблицы базы данных с объектами Java, а затем предоставляет различные **API** для выполнения различных типов операций над таблицами данных.

3). Какие важные преимущества дает использование Hibernate Framework?

Hibernate является одним из самых востребованных **ORM** фреймворков для Java. И вот почему:

1. Hibernate устраняет множество повторяющегося кода, который постоянно преследует разработчика при работе с JDBC. Скрывает от разработчика множество кода, необходимого для управления ресурсами и позволяет сосредоточиться на бизнес логике.
2. Hibernate поддерживает **XML** так же как и **JPA** аннотации, что позволяет сделать реализацию кода независимой.
3. Hibernate предоставляет собственный мощный язык запросов (**HQL**), который похож на SQL. Стоит отметить, что HQL **полностью объектно-ориентирован** и понимает такие принципы, как наследование, полиморфизм и ассоциации (связи).
4. Hibernate — широко распространенный open source проект. Благодаря этому доступны тысячи открытых статей, примеров, а так же документации по использованию фреймворка.
5. Hibernate легко интегрируется с другими Java EE фреймворками, например, **Spring Framework** поддерживает встроенную интеграцию с Hibernate.
6. Hibernate поддерживает **ленивую инициализацию** используя **proxy** объекты и выполняет запросы к базе данных только по необходимости.
7. Hibernate поддерживает разные **уровни cache**, а следовательно может повысить производительность.
8. Важно, что Hibernate может использовать чистый SQL, а значит поддерживает возможность **оптимизации** запросов и работы с любым сторонним БД и его фичами.

4). Какие преимущества Hibernate над JDBC?

Hibernate имеет ряд преимуществ перед JDBC API:

1. Hibernate удаляет множество повторяющегося кода из JDBC API, а следовательно его легче читать, писать и поддерживать.
2. Hibernate поддерживает наследование, ассоциации и коллекции, что не доступно в JDBC API.
3. Hibernate неявно использует управление **транзакциями**. Большинство запросов нельзя выполнить вне транзакции. При использовании JDBC API для управления транзакциями нужно явно использовать commit и rollback.
4. JDBC API throws SQLException, которое относится к проверяемым исключениям, а значит необходимо **постоянно** писать множество блоков try-catch. В большинстве случаев это не нужно для каждого вызова JDBC и используется для управления транзакциями. Hibernate оборачивает исключения JDBC через непроверяемые JDBCException или HibernateException, а значит нет необходимости проверять их в коде каждый раз. Встроенная поддержка управления транзакциями в Hibernate убирает блоки try-catch.
5. Hibernate Query Language (HQL) более объектно ориентированный и близкий к Java языку, чем SQL в JDBC.
6. Hibernate поддерживает кэширование, а запросы JDBC — нет, что может понизить производительность.
7. Конфигурация Hibernate позволяет использовать JDBC вроде соединения по типу JNDI DataSource для пула соединений. Это важная фича для энтерпрайз приложений, которая полностью отсутствует в JDBC API.
8. Hibernate поддерживает аннотации **JPA**, а значит код является переносимым на другие ORM фреймворки, реализующие стандарт, в то время как код JDBC сильно привязан к приложению.

5). Назовите некоторые важные интерфейсы Hibernate.

1. **SessionFactory (org.hibernate.SessionFactory)** — неизменяемый потокобезопасный объект с компилированным маппингом для одной базы данных. Необходимо инициализировать SessionFactory **всего один раз**. Экземпляр SessionFactory используется для получения объектов Session, которые используются для операций с базами данных.
2. **Session (org.hibernate.Session)** — однопоточный короткоживущий объект, который предоставляет связь между объектами приложения и базой данных. Он **оборачивает** JDBC java.sql.Connection и работает как фабрика для org.hibernate.Transaction. Разработчик должен открывать сессию по необходимости и закрывать ее сразу после использования. Экземпляр Session является интерфейсом между кодом в java приложении и hibernate framework и предоставляет методы для операций **CRUD**.
3. **Transaction (org.hibernate.Transaction)** — однопоточный короткоживущий объект, используемый для атомарных операций. Это абстракция приложения от основных JDBC или JTA транзакций. org.hibernate.Session может занимать несколько org.hibernate.Transaction в определенных случаях.

6). Что такое конфигурационный файл Hibernate?

Файл конфигурации Hibernate содержит в себе данные о базе данных и необходим для инициализации **SessionFactory**. В **.xml** файле необходимо указать вендора базы данных или JNDI ресурсы, а так же информацию об используемом диалекте, что поможет hibernate выбрать режим работы с конкретной базой данных.

7). Что такое Hibernate mapping file?

Файл отображения (**mapping file**) используется для связи entity бинов и колонок в таблице базы данных. В случаях, когда не используются аннотации JPA, файл отображения .xml может быть полезен (например при использовании сторонних библиотек).

8). Назовите некоторые важные аннотации, используемые для отображения в Hibernate.

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

1. **javax.persistence.Entity**: используется для указания класса как entity bean.
2. **javax.persistence.Table**: используется для определения имени таблицы из БД, которая будет отображаться на entity bean.
3. **javax.persistence.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. **javax.persistence.Id**: определяет primary key в entity bean.
5. **javax.persistence.EmbeddedId**: используется для определения составного ключа в бине.
6. **javax.persistence.Column**: определяет имя колонки из таблицы в базе данных.
7. **javax.persistence.GeneratedValue**: задает стратегию создания основных ключей. Используется в сочетании с javax.persistence.GenerationType enum.
8. **javax.persistence.OneToOne**: задает связь один-к-одному между двумя сущностными бинами. Соответственно есть другие аннотации OneToMany, ManyToOne и ManyToMany.
9. **org.hibernate.annotations.Cascade**: определяет каскадную связь между двумя entity бинами. Используется в связке с org.hibernate.annotations.CascadeType.
10. **javax.persistence.PrimaryKeyJoinColumn**: определяет внешний ключ для свойства. Используется вместе с org.hibernate.annotations.GenericGenerator и org.hibernate.annotations.Parameter.

9). Что вы знаете о Hibernate SessionFactory и как его сконфигурировать?

SessionFactory является фабрикой классов и используется для получения объектов session.
Фабрика сессий инициализируется на основе конфигурации Hibernate (например, через файл **hibernate.cfg.xml**) и предоставляет сессии для работы с базой данных.
Обычно в приложении имеется **только один** экземпляр SessionFactory и потоки, обслуживающие клиентские запросы, получают экземпляры session с помощью объекта SessionFactory.
Внутреннее состояние SessionFactory неизменно (**immutable**) и включает в себя все метаданные об **Object Relational Mapping** и задается при создании SessionFactory.
SessionFactory также предоставляет методы для **получения метаданных** класса и статистики, вроде данных о **втором уровне кэша**, выполняемых запросах и т.д.

10). Является ли Hibernate SessionFactory потокобезопасным?

Т.к. объект SessionFactory **immutable** (неизменяемый), **то да**, он потокобезопасный. Множество потоков может обращаться к одному объекту одновременно.

11). Как получить Hibernate Session и что это такое?

Объект Hibernate **Session** является связью между кодом java приложения и hibernate. Это **основной интерфейс** для выполнения операций с базой данных. Жизненный цикл объекта session связан с началом и окончанием транзакции. Сессия открывается в начале транзакции и закрывается по её завершении. Этот объект предоставляет методы для **CRUD** (create, read, update, delete) операций для объекта **персистентности**. С помощью этого экземпляра можно выполнять HQL, SQL запросы и задавать критерии выборки.

12). Является ли Hibernate Session потокобезопасным?

Объект Hibernate Session **не является потокобезопасным**. Каждый поток должен иметь свой собственный объект Session и закрывать его по окончанию.

13). В чем разница между openSession и getCurrentSession?

Hibernate SessionFactory **getCurrentSession()** возвращает сессию, связанную с контекстом. Но для того, чтобы это работало, нам нужно настроить его в конфигурационном файле hibernate. Так как этот объект session связан с контекстом hibernate, то отпадает необходимость к его закрытию. Объект session закрывается вместе с закрытием SessionFactory.

<property name="hibernate.current_session_context_class">thread</property>

Метод Hibernate SessionFactory **openSession() всегда создает новую сессию**. Мы должны обязательно контролировать **закрытие** объекта session по завершению всех операций с базой данных. Для многопоточной среды необходимо создавать новый объект session для каждого запроса. Существует еще один метод **openStatelessSession()**, который возвращает session **без поддержки состояния**. Такой объект **не реализует** первый уровень кэширования **и не** взаимодействует с вторым уровнем. Сюда же можно отнести игнорирование коллекций и некоторых обработчиков событий. Такие объекты могут быть полезны при загрузке больших объемов данных без удержания большого кол-ва информации в кэше.

14). Какая разница между методами Hibernate Session get() и load()?

get() и **load**() - это два разных метода для получения объектов из базы данных в Hibernate, и они имеют некоторые важные различия:

1) **get**():
   - Если объект не найден в базе данных, get() возвращает null.
   - Вызывая get(), Hibernate выполняет запрос к базе данных сразу же, чтобы найти объект и загрузить его в память.
   - Этот метод может использоваться, когда вы хотите получить объект из базы данных, и если он не существует, вам необходимо получить null в результате.

```
MyEntity entity = (MyEntity) session.get(MyEntity.class, id);
```

2) **load**():
   - Если объект не найден в базе данных, load() не возвращает null. Вместо этого, он создает прокси-объект (ленивую загрузку) и возвращает его. Прокси-объект не содержит реальных данных, и загрузка данных из базы данных происходит только тогда, когда к объекту обращаются (ленивая загрузка). Если объект существует, load() вернет его.
   - Этот метод может использоваться, когда вы уверены, что объект существует, и вы хотите использовать ленивую загрузку для оптимизации производительности.

```
MyEntity entity = (MyEntity) session.load(MyEntity.class, id);
```

Разница в поведении между get() и load() заключается в том, что get() возвращает null, если объект не найден, в то время как load() возвращает прокси-объект и выполняет ленивую загрузку. Выбор между этими методами зависит от требований вашего приложения и того, как вы хотите обрабатывать отсутствующие объекты.Нужно использовать метод get(), если необходимо удостовериться в наличии данных в БД.

15). Что вы знаете о кэшировании в Hibernate? Объясните понятие кэш первого уровня в Hibernate?

Hibernate использует кэширование, чтобы сделать наше приложение **быстрее**. Кэш Hibernate может быть очень полезным в получении высокой производительности приложения при правильном использовании.
Идея кэширования заключается в сокращении количества запросов к базе данных.
Кэш **первого уровня** Hibernate **связан с объектом Session и включен по умолчанию** и не существует никакого способа, чтобы его отключить. Однако Hibernate предоставляет методы, с помощью которых мы можем **удалить** выбранные объекты **из кэша** или полностью **очистить кэш**.
Любой объект закэшированный в session **не будет виден** другим объектам session. После закрытия объекта сессии все кэшированные объекты будут потеряны.

16). Как настроить кэш второго уровня в Hibernate с помощью EHCache?

EHCache является лучшим выбором для организации кэширования второго уровня в Hibernate. Для настройки второго уровня кэширования в Hibernate требуется выполнить несколько шагов.

- **Добавить зависимость hibernate-ehcache в проект**.

```
<dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-ehcache</artifactId>
      <version>4.3.5.Final</version>
</dependency>
```

- **Добавить несколько записей в конфигурационный файл Hibernate**.

```
<property name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>

<!-- For singleton factory -->
<!-- <property
name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory</property>
-->

<!-- enable second level cache and query cache -->
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
<property name="net.sf.ehcache.configurationResourceName">/myehcache.xml</property>
```

16). Продолжение

- **Создать файл конфигурации EHCache.**

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="ehcache.xsd" updateCheck="true"
   monitoring="autodetect" dynamicConfig="true">

   <diskStore path="java.io.tmpdir/ehcache" />

   <defaultCache maxEntriesLocalHeap="10000" eternal="false"
      timeToIdleSeconds="120" timeToLiveSeconds="120" diskSpoolBufferSizeMB="30"
      maxEntriesLocalDisk="10000000" diskExpiryThreadIntervalSeconds="120"
      memoryStoreEvictionPolicy="LRU" statistics="true">
      <persistence strategy="localTempSwap" />
   </defaultCache>

   <cache name="employee" maxEntriesLocalHeap="10000" eternal="false"
      timeToIdleSeconds="5" timeToLiveSeconds="10">
      <persistence strategy="localTempSwap" />
   </cache>

   <cache name="org.hibernate.cache.internal.StandardQueryCache"
      maxEntriesLocalHeap="5" eternal="false" timeToLiveSeconds="120">
      <persistence strategy="localTempSwap" />
   </cache>

   <cache name="org.hibernate.cache.spi.UpdateTimestampsCache"
      maxEntriesLocalHeap="5000" eternal="true">
      <persistence strategy="localTempSwap" />
   </cache>
</ehcache>
```

16). Продолжение

- **Использовать аннотацию @Cache и указание настройки стратегии кэширование над entity bean**.

```
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity
@Table(name = "ADDRESS")
@Cache(usage=CacheConcurrencyStrategy.READ_ONLY, region="employee")
public class Address {

}
```

17). Какие существуют различные состояния у entity bean?

1. **Transient**: состояние, при котором объект **никогда не был** связан с какой-либо сессией и не является персистентностью. Этот объект находится во временном состоянии. Объект в этом состоянии может стать персистентным при вызове метода save(), persist() или saveOrUpdate(). Объект персистентности может перейти в transient состоянии после вызова метода delete().
2. **Persistent**: когда объект **связан с** уникальной сессией он находится в состоянии persistent (персистентности). Любой экземпляр, возвращаемый методами get() или load() находится в состоянии persistent.
3. **Detached**: если объект **был** персистентным, но сейчас не связан с какой-либо сессией, то он находится в отвязанном (detached) состоянии. Такой объект можно сделать персистентным используя методы update(), saveOrUpdate(), lock() или replicate(). Состояния transient или detached так же могут перейти в состояние persistent как новый объект персистентности после вызова метода merge().

18). Как используется вызов метода Hibernate Session merge()?

Hibernate merge() может быть использован **для обновления** существующих значений, однако этот метод создает копию из переданного объекта сущности и возвращает его. Возвращаемый объект является частью контекста персистентности и отслеживает любые изменения, а переданный объект не отслеживается.

19). В чем разница между Hibernate save(), saveOrUpdate() и persist()?

Hibernate **save**() используется для сохранения сущности в базу данных. Проблема с использованием метода save() заключается в том, что он может быть вызван без транзакции. А следовательно если у нас имеется отображение нескольких объектов, то только первичный объект будет сохранен и мы получим несогласованные данные. Также save() немедленно возвращает сгенерированный идентификатор.

Hibernate **persist**() аналогичен save() с транзакцией. persist() не возвращает сгенерированный идентификатор сразу.

Hibernate **saveOrUpdate**() использует запрос для вставки или обновления, основываясь на предоставленных данных. Если данные уже присутствуют в базе данных, то **будет выполнен запрос обновления**. Метод saveOrUpdate() можно применять без транзакции, но это может привести к аналогичным проблемам, как и в случае с методом save().

20). Что произойдет, если будет отсутствовать конструктор без аргументов у Entity Bean?

Hibernate использует **рефлексию** для создания экземпляров Entity бинов при вызове методов **get**() или **load**(). Для этого используется метод **Class.newInstance()**, который **требует** наличия конструктора без параметров. Поэтому, в случае его отсутствия, вы получите ошибку **HibernateException**.

21). В чем разница между sorted collection и ordered collection? Какая из них лучше?

При использовании алгоритмов сортировки из Collection API для сортировки коллекции используется сортированный список (**sorted list**). Для маленьких коллекций это не приводит к излишнему расходу ресурсов, но на больших коллекциях это может привести к потере производительности и ошибкам OutOfMemory. Так же entity бины должны реализовывать интерфейс **Comparable** или **Comparator** для работы с сортированными коллекциями.
При использовании фреймворка Hibernate для загрузки данных из базы данных мы можем применить **Criteria API** и команду **order by** для получения отсортированного списка (**ordered list**). Ordered list является **лучшим** выбором к sorted list, т.к. он использует сортировку на уровне базы данных. Она быстрее и не может привести к утечке памяти. Пример запроса к БД для получения ordered list:

```
List<Employee> empList = session.createCriteria(Employee.class)
                        .addOrder(Order.desc("id")).list();
```

22). Какие типы коллекций в Hibernate вы знаете?

1. **Set**: Коллекция, представляющая собой множество уникальных элементов. В Hibernate, это часто используется для представления множественных связей "многие-ко-многим".
2. **List**: Упорядоченная коллекция элементов, которая может содержать дубликаты. Она часто используется для представления списков данных, например, список заказанных товаров в заказе.
3. **Map**: Коллекция, которая представляет собой отображение ключ-значение. Может быть использована для представления связей "один-ко-многим" или для хранения пар ключ-значение.
4. **Bag**: Это аналог List, но без гарантии порядка элементов. Она менее эффективна с точки зрения производительности, чем List или Set.
5. **Array**: Используется для хранения данных в виде массива.
6. **Primitive Collections**: Коллекции, специализированные для хранения примитивных типов данных (например, int, double).
7. **Sorted Collections**: Коллекции, которые автоматически сортируют элементы в заданном порядке (например, отсортированный Set или List).
8. **Identifier Bag**: Похожа на Bag, но также содержит идентификаторы объектов, что может быть полезно при выполнении дополнительных запросов.

23). Как реализованы Join'ы Hibernate?

Существует несколько способов реализовать связи в Hibernate.

- Использовать ассоциации, такие как **one-to-one**, **one-to-many**, **many-to-many**.
- Использовать в HQL запросе команду JOIN. Существует другая форма <<join fetch>>, позволяющая загружать данные немедленно (**не lazy**).
- Использовать чистый SQL запрос с командой join.

24). Почему мы не должны делать Entity class как final?

Hibernate использует **прокси классы** для ленивой загрузки данных (т.е. по необходимости, а не сразу). Это достигается с помощью расширения entity bean и, следовательно, если бы он был final, то это было бы невозможно. Ленивая загрузка данных во многих случаях повышает производительность, а следовательно важна.

25). Что вы знаете о HQL и какие его преимущества?

HQL (Hibernate Query Language) - это язык запросов, разработанный для выполнения запросов к данным в Hibernate, фреймворке объектно-реляционного отображения (ORM) для Java. HQL аналогичен **SQL** (Structured Query Language), но использует объекты и свойства Java-классов вместо таблиц и столбцов базы данных. Вот некоторые основные аспекты HQL и его преимущества:

1. **Объектно-Ориентированный Подход**: HQL позволяет разработчикам создавать запросы, используя объектную модель данных, а не таблицы и столбцы. Это делает код более читаемым и обеспечивает более натуральное взаимодействие с данными.
2. **Портабельность**: HQL позволяет писать запросы, которые могут быть перенесены между различными базами данных без изменения кода. Hibernate обрабатывает различия в SQL-диалектах различных СУБД.
3. **Использование Отображений Объектов**: HQL позволяет работать с объектами и их ассоциациями, а не требует прямого написания SQL-кода. Например, вы можете обращаться к свойствам объектов в HQL-запросах, а не к столбцам базы данных.
4. **Поддержка Явных и Неявных Join'ов**: HQL поддерживает как явные, так и неявные (автоматически генерируемые) объединения, что делает запросы более гибкими.
5. **Функции и Агрегатные Функции**: HQL предоставляет множество встроенных функций и агрегатных функций, аналогичных SQL, которые можно использовать в запросах.
6. **Поддержка Параметров**: Вы можете передавать параметры в HQL-запросы, что делает их параметризованными и улучшает безопасность и производительность.
7. **Легкая Интеграция с Hibernate**: HQL является частью Hibernate и интегрируется непосредственно с маппингом объектов Hibernate, что облегчает работу с данными и уменьшает необходимость писать сложный SQL-код.

26). Что такое Query Cache в Hibernate?

Hibernate реализует область кэша для запросов **resultset**, который тесно взаимодействует с кэшем второго уровня Hibernate. Для подключения этой дополнительной функции требуется несколько дополнительных шагов в коде. Query Cache полезны только для часто выполняющихся запросов с повторяющимися параметрами. Для начала необходимо добавить эту запись в файле конфигурации Hibernate:

```
<property name="hibernate.cache.use_query_cache">true</property>
```

Уже внутри кода приложения для запроса применяется метод **setCacheable**(**true**), как показано ниже:

```
Query query = session.createQuery("from Employee");
query.setCacheable(true);
query.setCacheRegion("ALL_EMP");
```

27). Можем ли мы выполнить нативный запрос SQL (sql native) в Hibernate?

С помощью использования SQLQuery можно выполнять чистый запрос SQL. В общем случае это не рекомендуется, т.к. вы потеряете все преимущества HQL (ассоциации, кэширование). Выполнить можно примерно так:

```
Transaction tx = session.beginTransaction();

SQLQuery query = session.createSQLQuery("select emp_id, emp_name, emp_salary from Employee");

List<Object[]> rows = query.list();
for(Object[] row : rows){
    Employee emp = new Employee();
    emp.setId(Long.parseLong(row[0].toString()));
    emp.setName(row[1].toString());
    emp.setSalary(Double.parseDouble(row[2].toString()));
    System.out.println(emp);
}
```

28). Назовите преимущества поддержки нативного sql в Hibernate.

Использование нативного SQL может быть необходимо при выполнении запросов к некоторым базам данных, которые могут не поддерживаться в Hibernate. Примером может служить некоторые специфичные запросы и «фишки» при работе с БД от Oracle.

29). Расскажите о преимуществах использования Hibernate Criteria API.

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.
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()**.

30). Как логировать созданные Hibernate SQL запросы в лог-файлы?

Для логирования SQL запросов, созданных Hibernate, вы можете использовать инструменты и настройки логирования, предоставляемые Hibernate, в сочетании с популярными библиотеками логирования, такими как Log4j, SLF4J или Java Util Logging. Вот как это можно сделать:

- **Выбор Библиотеки Логирования**: Выберите библиотеку логирования, которую вы хотите использовать, и добавьте ее зависимость в ваш проект. Например, вы можете использовать Log4j2, добавив его зависимость в файл **pom.xml** вашего проекта

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.x.x</version> <!-- Замените на актуальную версию Log4j2 -->
</dependency>
```

- **Настройка Логирования Hibernate**: В файле конфигурации Hibernate (например, hibernate.cfg.xml или через Java классы конфигурации) добавьте следующие настройки:

```
<property name="hibernate.show_sql">true</property>
<property name="hibernate.format_sql">true</property>
```

hibernate.show_sql: Установите это свойство в true, чтобы Hibernate показывал созданные SQL запросы в консоли.
hibernate.format_sql: Установите это свойство в true, чтобы запросы были красиво форматированы.

30).
Продолжение

- **Настройка Логирования в Log4j (или Другой Логгер)**: Создайте файл конфигурации для вашего логгера (например, log4j2.xml для Log4j2) и настройте логирование Hibernate. В примере для Log4j2 это может выглядеть так:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Logger name="org.hibernate.SQL" level="DEBUG"/>
    <Logger name="org.hibernate.type.descriptor.sql.BasicBinder" level="TRACE"/>
    <Root level="error">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

Этот файл конфигурации настраивает логирование Hibernate SQL запросов на уровне DEBUG и выше.

- **Запустите Приложение**: Теперь, при запуске вашего приложения, Hibernate будет логировать созданные SQL запросы в вашем выбранном лог-файле или консоли в зависимости от настроек логгера.

31). Что вы знаете о Hibernate прокси и как это помогает в ленивой загрузке (lazy load)?

Hibernate **Proxy** - это мощный механизм, который позволяет оптимизировать загрузку данных и улучшить производительность приложения. Он позволяет вам работать с ассоциациями как с обычными объектами, но при этом загружать данные только по мере необходимости(**lazy loading**).
Ленивая загрузка позволяет отложить загрузку связанных данных до тех пор, пока они действительно не понадобятся, что улучшает производительность приложения.

Вот как Hibernate Proxy работает и как он помогает в ленивой загрузке:

1. Генерация Прокси-Объектов: При загрузке объекта с ассоциацией, помеченной как ленивая (например, @ManyToOne или @OneToMany с опцией FetchType.LAZY), Hibernate не загружает связанный объект сразу. Вместо этого он создает специальный прокси-объект, который является подклассом вашего класса и наследует его методы.
2. Инициализация Прокси: Когда вы пытаетесь получить доступ к данным в ассоциированном объекте (например, вызываете метод get() на ассоциированном поле), Hibernate проверяет, инициализирован ли уже прокси. Если нет, то он инициализирует его, выполнив запрос к базе данных для загрузки данных связанного объекта.
3. Преимущества Ленивой Загрузки:
    - Экономия ресурсов: Ленивая загрузка позволяет избегать избыточной загрузки данных, которые могут не понадобиться.
    - Улучшенная производительность: Меньше запросов к базе данных при загрузке объектов.
    - Гибкость: Вы можете выбирать, какие ассоциации загружать лениво, а какие - сразу.
4. Типы Прокси: Hibernate поддерживает два типа прокси-объектов:
    - **Runtime Proxy**: Этот тип прокси создается во время выполнения с использованием байткода. Он обеспечивает более высокую производительность, но требует наличия библиотеки байткода (например, CGLIB).
    - **Javassist Proxy**: Этот тип прокси создается с использованием библиотеки Javassist. Он менее производительный, но не требует наличия библиотеки байткода.

32). Как реализованы отношения в Hibernate?

Hibernate предоставляет различные способы для определения и реализации отношений между сущностями. Вы можете использовать JPA аннотации или конфигурацию XML файлов для описания этих отношений. Вот как это можно сделать:

1) One-to-One (Один-к-Одному):

- С JPA аннотациями: Используйте аннотации @OneToOne на поле или методе, чтобы определить отношение. Например:

```
@Entity
public class Person {
@OneToOne
private Address address;
    // ...
}
```

- С использованием XML: Определите соответствующий элемент <one-to-one> в вашем файле конфигурации XML.

2) One-to-Many (Один-к-Многим):

- С JPA аннотациями: Используйте аннотации @OneToMany на поле или методе в родительской сущности, чтобы указать отношение. Например:

```
@Entity
public class Department {
    @OneToMany(mappedBy = "department")
    private List<Employee> employees;
    // ...
}
```

- С использованием XML: Определите соответствующий элемент <one-to-many>.

32).
Продолжение

3) Many-to-Many (Многие-ко-Многим):

- С JPA аннотациями: Используйте аннотации @ManyToMany на поле или методе в обеих сущностях, чтобы определить отношение. Например:

```
@Entity
public class Student {
    @ManyToMany
    @JoinTable(name = "student_course",
            joinColumns = @JoinColumn(name = "student_id"),
            inverseJoinColumns = @JoinColumn(name = "course_id"))
    private List<Course> courses;
    // ...
}
```

- С использованием XML: Определите соответствующий элемент <many-to-many> и настройте таблицу промежуточного соединения.

33). Какие типы менеджмента транзакций поддерживаются в Hibernate?

Hibernate поддерживает два основных типа менеджмента транзакций:

1) Управление транзакциями с помощью **API** языка программирования (**Programmatic Transaction Management**): Этот метод позволяет программисту явно управлять транзакциями с использованием API, предоставляемого Hibernate или платформой, на которой работает ваше приложение. Программист может начать, зафиксировать или откатить транзакцию вручную с помощью соответствующих методов. Этот метод предоставляет максимальную гибкость, но может привести к более многословному коду.

Пример использования управления транзакциями с помощью API:

```
Session session = sessionFactory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    // Выполнение операций с базой данных
    tx.commit();
} catch (Exception e) {
    if (tx != null) {
        tx.rollback();
    }
    e.printStackTrace();
} finally {
    session.close();
}
```

33). Продолжение

2) Управление транзакциями с помощью декларативного подхода (**Declarative Transaction Management**): В этом методе управление транзакциями выносится за пределы кода приложения и настраивается с помощью конфигурации. Обычно этот метод используется в комбинации с контейнером Spring или другой платформой, которая предоставляет декларативную конфигурацию транзакций. Транзакции объявляются как аспекты, и их поведение настраивается в конфигурационных файлах. Этот метод обеспечивает сокращение кода и упрощение управления транзакциями.

Пример декларативного управления транзакциями в Spring с использованием аннотаций:

```
@Service
public class MyService {
    @Autowired
    private MyRepository repository;

    @Transactional
    public void doSomething() {
        // Выполнение операций с базой данных
    }
}
```

Оба метода имеют свои преимущества и подходят для разных сценариев. Выбор метода зависит от требований вашего приложения и вашей предпочтительной архитектуры.

34). Что такое каскадные связи (обновления) и какие каскадные типы есть в Hibernate?

Если у нас имеются зависимости между сущностями (entities), то нам необходимо определить как различные операции будут влиять на другую сущность. Это реализуется с помощью каскадных связей (или обновлений). Вот пример кода с использованием аннотации **@Cascade**:

```
import org.hibernate.annotations.Cascade;

@Entity
@Table(name = "EMPLOYEE")
public class Employee {

@OneToOne(mappedBy = "employee")
@Cascade(value = org.hibernate.annotations.CascadeType.ALL)
private Address address;

}
```

Обратите внимание, что есть некоторые различия между enum CascadeType в Hibernate и в JPA. Поэтому обращайте внимание какой пакет вы импортируете при использовании аннотации и константы типа. Наиболее часто используемые CascadeType перечисления описаны ниже.

1. **None**: без Cascading. Формально это не тип, но если мы не указали каскадной связи, то никакая операция для родителя не будет иметь эффекта для ребенка.
2. **ALL**: Cascades save, delete, update, evict, lock, replicate, merge, persist. В общем — всё.
3. **SAVE_UPDATE**: Cascades save и update. Доступно только для hibernate.
4. **DELETE**: передает в Hibernate native **DELETE** действие. Только для hibernate.
5. **DETATCH**, **MERGE**, **PERSIST**, **REFRESH** и **REMOVE** – для простых операций.
6. **LOCK**: передает в Hibernate native **LOCK** действие.
7. **REPLICATE**: передает в Hibernate native **REPLICATE** действие.

35). Что вы знаете о классе HibernateTemplate?

Spring Framework предоставляет различные подходы для интеграции с Hibernate. Тем не менее, наиболее часто используется подход, использующий HibernateTemplate. Есть две основные причины:

- Класс скрывает детали управления сессиями и транзакциями.
- Предоставляет подход основанный на шаблонах

HibernateTemplate класс скрывает трудности управления сессиями и транзакциями при использовании Hibernate для доступа к данным. Нужно только инициализировать HibernateTemplate путем передачи экземпляра SessionFactory. Spring Framework берет на себя беспокойство за детали связанные с сессиями и транзакциями. Это помогает устранить инфраструктурный код, который может вносить суматоху при увеличении сложности.

HibernateTemplate, так же как и JdbcTemplate, предоставляет шаблонный подход для доступа к данным. Когда вы используете HibernateTemplate, вы будете работать с callbacks. Обратные вызовы — это единственный механизм в шаблонном подходе, который уведомляет шаблон запускать нужную задачу. Преимущество наличия обратного вызова в том, что там только одна точка входа в слой доступа к данным. И эта точка входа определяется шаблоном, в этом случае HibernateTemplate.

В комментариях дополнили, что использование HibernateTemplate не является рекомендуемым. Вместо использования HibernateTemplate из пакета org.springframework.orm рекомендуется использовать декларативный подход (@Transactional). Таким образом фреймворк сам позаботится об операциях **open, commit, close, flush.**

36). Какие паттерны применяются в Hibernate?

- Domain Model Pattern – объектная модель предметной области, включающая в себя как поведение так и данные.
- Data Mapper – слой мапперов (Mappers), который передает данные между объектами и базой данных, сохраняя их независимыми друг от друга и себя.
- Proxy Pattern — применяется для ленивой загрузки.
- Factory pattern — используется в SessionFactory

37). Расскажите о Hibernate Validator Framework.

Проверка данных является неотъемлемой частью любого приложения. Hibernate Validator обеспечивает эталонную реализацию двух спецификаций **JSR-303** и **JSR-349** применяемых в Java. Для настройки валидации в Hibernate необходимо сделать следующие шаги.

- **Добавить hibernate validation зависимости в проект.**

```
<dependency>
   <groupId>javax.validation</groupId>
   <artifactId>validation-api</artifactId>
   <version>1.1.0.Final</version>
</dependency>
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-validator</artifactId>
   <version>5.1.1.Final</version>
</dependency>
```

- **Так же требуются зависимости из JSR 341, реализующие Unified Expression Language для обработки динамических выражений и сообщений о нарушении ограничений.**

37). Продолжение

```
<dependency>
   <groupId>javax.el</groupId>
   <artifactId>javax.el-api</artifactId>
   <version>2.2.4</version>
</dependency>
<dependency>
   <groupId>org.glassfish.web</groupId>
   <artifactId>javax.el</artifactId>
   <version>2.2.4</version>
</dependency>
```

- **Использовать необходимые аннотации в бинах.**

```
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.CreditCardNumber;
import org.hibernate.validator.constraints.Email;

public class Employee {

   @Min(value=1, groups=EmpIdCheck.class)
   private int id;

   @NotNull(message="Name cannot be null")
   @Size(min=5, max=30)
   private String name;

   @Email
   private String email;

   @CreditCardNumber
   private String creditCardNumber;
....
```

38). Какие преимущества дает использование плагина Hibernate Tools Eclipse?

Плагин Hibernate Tools упрощает настройку маппинга, конфигурационного файла. Упрощает работы с файлами свойств или xml тегами. Помогает минимизировать ошибки написания кода.

39). Чем отличается Lazy от Eager в Hibernate?

- **Eager Loading** — стратегия загрузки, при которой подгрузка связанных сущностей происходит сразу. Для применения необходимо в аннотацию отношения (@OneToOne, @ManyToOne, @OneToMany, @ManyToMany) передать fetch = FetchType.EAGER. Используется по умолчанию для отношений @OneToOne и @ManyToOne.
- **Lazy Loading** — стратегия загрузки, при которой подгрузка связанных сущностей откладывается как можно дольше. Чтобы задать такое поведение, нужно в аннотацию отношения (@OneToOne, @ManyToOne, @OneToMany, @ManyToMany) передать fetch = FetchType.LAZY. Используется по умолчанию для отношений @OneToMany, @ManyToMany. До момента загрузки используется proxy-объект, вместо реального. Если обратиться к такому LAZY-полю после закрытия сессии Hibernate, то получим **LazyInitializationException**.

40). Что такое 'проблема N+1 запроса' при использовании Hibernate? Когда возникает? Как решить? Как обнаружить?

Проблема N+1 запроса (или проблема N+1 select) - это ситуация, которая возникает при использовании ORM-фреймворка, такого как Hibernate, когда при загрузке сущностей и связанных с ними объектов выполняется гораздо больше запросов к базе данных, чем ожидалось. Это может существенно ухудшить производительность приложения и привести к ненужной нагрузке на базу данных.
Пример ситуации, в которой возникает проблема N+1 запроса:

1. Пусть у вас есть сущность **Author**, представляющая авторов книг, и сущность **Book**, представляющая книги.
2. Существует отношение между **Author** и **Book**, где один автор может иметь много книг (one-to-many).
3. Вы хотите загрузить всех авторов и их книги.

Если вы используете стандартный способ загрузки данных с помощью Hibernate, то для каждого автора будет выполнен запрос к базе данных для загрузки его книг. То есть, сначала выполняется запрос для выбора всех авторов (это один запрос), а затем для каждого автора выполняется отдельный запрос для выбора его книг. Таким образом, если у вас есть N авторов, будет выполнено N+1 запросов к базе данных (1 запрос для выбора авторов и по одному запросу для каждого автора для выбора его книг).

40).
Продолжение

Для решения проблемы N+1 запроса в Hibernate можно использовать следующие подходы:

1. **Eager Loading** (Жадная загрузка): Вместо ленивой загрузки (по умолчанию) можно использовать жадную загрузку, чтобы загрузить все связанные объекты сразу. Для этого используется аннотация @OneToMany(fetch = FetchType.EAGER) (или аналогичный XML-конфигурационный параметр) над связью. Однако этот подход может загрузить слишком много данных, если они необходимы не всегда.
2. **Join Fetch**: Можно использовать ключевое слово JOIN FETCH в HQL или JPQL запросах для выборки связанных объектов одним запросом. Это позволяет сократить количество запросов, но также может привести к дублированию данных в результатах запроса.
3. **Batch Fetching**: Hibernate поддерживает пакетную загрузку (batch fetching), которая позволяет выполнить один запрос для загрузки коллекции объектов. Это особенно полезно, когда у вас есть связь многие-к-одному (one-to-many), и вы хотите загрузить все связанные объекты для нескольких родительских объектов одним запросом.
4. **Second-Level Cache**: Использование вторичного кэша Hibernate (например, Ehcache или Infinispan) может уменьшить количество запросов к базе данных, храня часто используемые данные в памяти.

Для обнаружения проблемы N+1 запроса можно воспользоваться мониторингом базы данных, логированием SQL-запросов или использовать специализированные инструменты для профилирования приложений. Часто такие инструменты могут выявить множество однотипных запросов к базе данных, что является признаком проблемы N+1 запроса.

41). Как описать составной ключ при использовании Hibernate?

Для описания составного ключа (**composite key**) при использовании Hibernate можно воспользоваться аннотациями или XML-конфигурацией. Сначала определите класс, который будет представлять составной ключ. Затем укажите его в сущности (entity) как поле с помощью аннотации **@EmbeddedId** (для аннотаций) или <composite-id> (для XML-конфигурации).
Вот пример того, как описать составной ключ с использованием аннотаций:

- Создайте класс, представляющий составной ключ. Этот класс должен реализовывать интерфейс **Serializable** и переопределить методы **equals**() и **hashCode**(). Например:

```
import java.io.Serializable;
import java.util.Objects;

public class MyCompositeKey implements Serializable {
    private Long keyPart1;
    private String keyPart2;

    // Геттеры и сеттеры для keyPart1 и keyPart2

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        MyCompositeKey that = (MyCompositeKey) o;
        return Objects.equals(keyPart1, that.keyPart1) &&
                Objects.equals(keyPart2, that.keyPart2);
    }

    @Override
    public int hashCode() {
        return Objects.hash(keyPart1, keyPart2);
    }
}
```

41).
Продолжение

- В сущности, которая будет использовать составной ключ, создайте поле с типом вашего составного ключа и пометьте его аннотацией @EmbeddedId:

```
@Entity
public class MyEntity {
    @EmbeddedId
    private MyCompositeKey id;

    // Другие поля сущности

    // Геттеры и сеттеры для id и других полей
}
```

Теперь Hibernate знает, что **MyCompositeKey** представляет составной ключ для сущности **MyEntity**. Вы можете использовать **MyCompositeKey** в качестве ключа для поиска сущностей или устанавливать его значения при сохранении новых сущностей. Помните, что составные ключи могут использоваться только сущностями, которые используются в режиме чтения и записи (можно модифицировать). Они не могут быть использованы в сущностях, которые только для чтения (read-only).

42). Как можно отобразить наследование на БД с помощью JPA (Hibernate)?

Есть 4 способа отобразить наследование на БД с помощью JPA (Hibernate):

- **MappedSuperclass** — поля родителя содержатся в каждой таблице для каждого дочернего класса. Базовый класс отдельной таблицы не имеет. На базовый класс навешиваем **@MappedSuperClass**, а вот на дочерние @Entity. Если в таблице потомка поле родителя называется не так, как указано в родительском классе, то его нужно смаппить с помощью аннотации **@AttributeOverride** в классе этого потомка. Родитель не может участвовать в ассоциации. При полиморфных запросах у нас будут отдельные запросы для каждой таблицы.
- **Single table** — вся иерархия классов в одной таблице. Чтобы различать классы, необходимо добавить колонку-дискриминатор. В данной стратегии на родительский @Entity-класс навешивается @Inheritance(strategy = InheritanceType.SINGLE_TABLE) и @DiscriminatorColumn(name = "YOUR_DISCRIMINATOR_COLUMN_NAME") (по умолчанию имя колонки DTYPE и тип VARCHAR). В каждом подклассе указываем @DiscriminatorValue("ThisChildName") со значением, которое будет храниться в колонке-дискриминаторе для данного класса. Если нет возможности добавить колонку, то можно использовать аннотацию **@DiscriminatorFormula**, в которой указать выражение CASE...WHEN — это не по JPA, фишка Hibernate. Денормализация. Простые запросы к одной таблице. Возможное нарушение целостности — столбцы подклассов могут содержать NULL.

42).
Продолжение

- **Joined table** — отдельные таблицы для всех классов иерархии, включая родителя. В каждой таблице только свои поля, а в дочерних добавляется внешний (он же первичный) ключ для связи с родительской таблицей. В @Entity-класс родителя добавляем @Inheritance(strategy = InheritanceType.JOINED). Для полиморфных запросов используются JOIN, а также выражение CASE...WHEN, вычисляющее значение поля _clazz, которое заполняется литералами (0 (родитель), 1, 2 и т.д.) и помогает Hibernate определить какого класса будет экземпляр.
- **Table per class** — также как и в MappedSuperclass, имеем отдельные таблицы для каждого подкласса. Базовый класс отдельной таблицы не имеет. По **спецификации** JPA 2.2 (раздел 2.12) данная стратегия является опциональной, но в Hibernate реализована, поэтому продолжим. В данном случае на базовый класс мы навешиваем @Entity и @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS). Поле первичного ключа (@Id) обязательно для родительского класса. Также аннотация @AttributeOverride в этой стратегии не работает — называйте родительские поля в таблицах сразу единообразно. Полиморфный запрос будет использовать UNION для объединения таблиц. Чтобы различить при создании экземпляров подклассы, Hibernate добавляет поле _clazz в запросы, содержащие литералы (1, 2 и т.д.). А одинаковый набор столбцов для объединения добирается как NULL AS some_field. Родитель может участвовать в ассоциации с другими сущностями.

43). Что такое диалект?

Диалект — это набор файлов кода или иногда один файл, который определяет процесс подключения базы данных к классу Java. Диалект в Hibernate играет роль понимания связи, происходящей с базовой базой данных. Всякий раз, когда изменяется базовая база данных, все, что вам нужно изменить в конфигурации Hibernate, — это диалект и учетные данные базы данных. Это верно, пока код использует HQL-запросы.

44). Как Hibernate создает соединение с базой данных?

Hibernate читает настроенный диалект, чтобы решить, какой драйвер использовать. Hibernate поставляется в комплекте с jar-файлами драйверов баз данных. Эти банки аналогичны тем, которые используются для подключения к базе данных с использованием JDBC. На основе диалекта Hibernate динамически регистрирует соответствующий класс драйверов и использует URL-адрес и учетные данные для подключения к базе данных с использованием JDBC в бэкэнде.

45). Какая аннотация используется для объявления класса как сущность ?

Аннотация @Entity используется для объявления класса как объекта. Простой пример показан ниже.

```
@Entity
@Table(name="users")
public class User{
String username;
String password;
}
```

46). Как мне указать имя таблицы, связанной с объектом, используя аннотацию?

```
@Entity
@Table(name="users")
public class User{
String username;
String password;
}
```

Как видно из приведенного выше кода, аннотация **@Table** используется для указания имени таблицы базы данных, связанной с объектом. Для этого объекта требуется обязательное name атрибута, которое указывает имя таблицы, как в базе данных.

47). Как переменная в сущности соединяется со столбцом базы данных?

По умолчанию Hibernate ищет имена столбцов, соответствующие именам переменных в классе. Однако также возможно указывать разные имена переменных и связывать их с соответствующими столбцами в базе данных.

48). Как указать другое имя столбца для отображения переменных?

Аннотация **@Column** используется для определения имени столбца, связанного с переменной. В отсутствие этой аннотации Hibernate предварительно компилирует отображение переменной, сопоставленной со столбцом с тем же именем. Пример использования этой аннотации показан ниже:

```
@Entity
@Table(name="users")
public class User{
@Column(name="user_name")
String username;
String password;
}
```

Атрибут **name** является обязательным атрибутом для указания имени столбца, отличного от имени переменной. Из приведенного выше кода можно понять, что столбец **user_name** связан с переменной **username**

49). Как мы указываем переменную, которая будет первичным ключом для таблицы?

Hibernate может создавать таблицы базы данных для приложения непосредственно на основе отображений, представленных в коде Java. В таком случае Hibernate требует знать, какие столбцы должны быть первичными ключами. Это можно настроить с помощью аннотации **@Id** . Hibernate не только заботится о создании этого столбца как столбца первичного ключа, но также проверяет свое уникальное ограничение при каждой вставке и обновлении базы данных.

50). Как мы определяем логику генерации значения первичного ключа?

Значения первичного ключа могут быть сгенерированы различными способами в зависимости от базы данных. Например, в базе данных MySQL первичные ключи могут быть сгенерированы с использованием алгоритма автоинкрементации, в то время как в базе данных Oracle вам необходимо создать последовательность и использовать ее для автоматического увеличения значения для первичного ключа. Эти методы генерации могут быть указаны с помощью приведенного ниже кода аннотации.

```
@Entity
@Table(name="users")
public class User{

@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
int userid;

@Column(name="user_name")
String username;

String password;
}
```

Столбец идентификатора пользователя здесь определен как первичный ключ, автоматически сгенерированный с использованием стратегии идентификации. Возможные значения для strategy включают в себя:

- GenerationType.AUTO
- GenerationType.IDENTITY
- GenerationType.SEQUENCE
- GenerationType.TABLE

51). Как вы настраиваете диалект в hibernate.cfg.xml?

Конфигурация диалекта в xml включает определение свойства с именем hibernate.dialect. Пример XML-тега для определения диалекта показан ниже:

```
<property name="org.hibernate.dialect.MySQLDialect">
org.hibernate.dialect.MySQLDialect
</property>
```

52). Как настроить URL базы данных и учетные данные в hibernate.cfg.xml?

Hibernate Framework может быть настроен с использованием различных значений свойств. Пример URL базы данных конфигурации и учетных данных приведен ниже.

<property name = "hibernate.connection.url">jdbc:mysql://localhost/mydb</property>

<property name = "hibernate.connection.username">root</property>

<property name = "hibernate.connection.password">password</property>

При выполнении приложения Java платформа Hibernate предварительно компилирует код для подключения к базе данных и создает пул подключений, чтобы уменьшить накладные расходы на подключение к базе данных во время выполнения запросов.

53). Как настроить размер пула соединений?

Размер пула соединений в Hibernate имеет два значения — минимальный размер пула и максимальный размер пула. Эти размеры можно настроить с помощью свойств **hibernate.c3p0.min_size** и **hibernate.c3p0.max_size** . Эти свойства можно настроить так же, как показано выше для учетных данных базы данных.

54). Как мы совершаем транзакцию в Hibernate?

Объект транзакции в Hibernate может быть зафиксирован или откатан. Для выполнения этого действия мы используем приведенный ниже код.

```
tx = Session.beginTransaction();
...
...
...
//Do something with transaction
...
...
...
tx.commit();
```

Как видно, вызов функции **tx.commit()** выполняет задачу транзакции в базе данных. Для отката процедура остается прежней. Все, что вам нужно сделать, это изменить вызов функции на **tx.rollback()** .

55). Можно ли подключить несколько баз данных в одном приложении Java с помощью Hibernate?

Да. Практически можно подключить одно приложение Java к нескольким базам данных, используя два отдельных файла конфигурации Hibernate и две отдельные SessionFactory. Эти файлы конфигурации содержат различные конфигурации диалектов, относящихся к соответствующей базе данных. Объекты исключительно отображаются в соответствующую конфигурацию базы данных. Таким образом, с двумя разными параллельными объектами SessionFactory можно подключить несколько баз данных.

56). Поддерживает ли Hibernate полиморфизм?

Да. Hibernate по своей природе поддерживает полиморфизм. Hibernate классы в основном поддерживают свои операции посредством самого полиморфизма.

57). Сколько сессий Hibernate существует в любой момент времени в приложении?

Hibernate сессия является общим объектом. В любой момент времени существует только один общий объект сеанса, который помогает в управлении транзакциями и получении соединений из пула соединений. Следует отметить, что это верно только при использовании одной конфигурации базы данных. В случае нескольких конфигураций базы данных Hibernate создаст отдельный объект сеанса для поддержки сопоставления и транзакций для другой базы данных.

58). Какие изоляции транзакций есть в Hibernate?

Hibernate поддерживает четыре уровня изоляции транзакций, которые могут быть заданы с помощью аннотаций или XML-конфигурации:

- **READ_UNCOMMITTED** - это наименьший уровень изоляции, который позволяет одной транзакции видеть изменения, внесенные другой транзакцией до их фиксации. Этот уровень может привести к "грязному чтению", когда транзакция видит данные, которые могут быть отменены.
- **READ_COMMITTED** - это уровень изоляции по умолчанию в Hibernate. Он гарантирует, что транзакция видит только изменения, зафиксированные другими транзакциями. Это предотвращает "грязное чтение", но может привести к "неповторяемому чтению" при повторном чтении данных, которые были изменены другой транзакцией между двумя чтениями.
- **REPEATABLE_READ** - это уровень изоляции, который гарантирует, что транзакция видит одни и те же данные при повторном чтении в рамках той же самой транзакции. Транзакция не видит изменения, внесенные другими транзакциями после начала текущей транзакции.
- **SERIALIZABLE** - это наивысший уровень изоляции, который гарантирует, что транзакция видит данные в том же самом состоянии, что и при начале транзакции. Он предотвращает "грязное чтение", "неповторяемое чтение" и "фантомное чтение", но может привести к замедлению производительности.

Выбор уровня изоляции зависит от требований к приложению и конкретных сценариев использования.

59). Чем отличаются JPA и Hibernate?

В начале стоит отметить, что **JPA (Java Persistence API)** это **спецификация**, а Hibernate **это реализация** этой спецификации. Спецификация JPA предоставляет стандарты для ORM (Object-Relational Mapping) технологий в Java. Это означает, что JPA определяет набор правил и руководств по тому, как должен работать ORM-инструмент.

60). Как интегрировать Hibernate и Spring?

Лучше всего прочитать о настройках на сайтах фреймворков для текущей версии. Оба фреймворка поддерживают интеграцию из коробки и в общем настройка их взаимодействия не составляет труда. Общие шаги выглядят следующим образом.

- Добавить зависимости для hibernate-entitymanager, hibernate-core и spring-orm.
- Создать классы модели и передать реализации DAO операции над базой данных. Важно, что DAO классы используют SessionFactory, который внедряется в конфигурации бинов Spring.
- Настроить конфигурационный файл Spring (смотрите в офф. документации).
- Дополнительно появляется возможность использовать аннотацию @Transactional и перестать беспокоиться об управлении транзакцией Hibernate.