Ответы на вопросы на собеседование Java Persistence API (JPA) (часть 1).

Vasyl K 8:01:00 Добавить комментарий

- **ЧТО ТАКОЕ JPA?**

JPA – это технология, обеспечивающая объектно-реляционное отображение простых JAVA объектов и предоставляющая API для сохранения, получения и управления такими объектами.
JPA – это спецификация (документ, утвержденный как стандарт, описывающий все аспекты технологии), часть EJB3 спецификации.
Сам JPA не умеет ни сохранять, ни управлять объектами, JPA только определяет правила игры: как что-то будет действовать. JPA также определяет интерфейсы, которые должны будут быть реализованы провайдерами. Плюс к этому JPA определяет правила о том, как должны описываться метаданные отображения и о том, как должны работать провайдеры. Дальше, каждый провайдер, реализуя JPA определяет получение, сохранение и управление объектами. У каждого провайдера реализация разная.

- Реализации JPA:
- Hibernate
- Oracle TopLink
- Apache OpenJPA

- **ИЗ ЧЕГО СОСТОИТ JPA?**

JPA состоит из трех основных пунктов:

- API – интерфейсы в пакете javax.persistance. Набор интерфейсов, которые позволяют организовать взаимодействие с ORM провайдером.
- JPQL – объектный язык запросов. Очень похож на SQL, но запросы выполняются к объектам.
- Metadata – аннотации над объектами. Набор аннотаций, которыми мы описываем метаданные отображения. Тогда уже JPA знает какой объект в какую таблицу нужно

сохранить. Метаданные можно описывать двумя способами: XML-файлом или через аннотации.

- **В ЧЕМ ЕЁ ОТЛИЧИЕ JPA ОТ HIBERNATE?**

Hibernate одна из самых популярных открытых реализаций последней версии спецификации (JPA 2.1). То есть JPA только описывает правила и API, а Hibernate реализует эти описания, впрочем у Hibernate (как и у многих других реализаций JPA) есть дополнительные возможности, не описанные в JPA (и не переносимые на другие реализации JPA).

- **В ЧЕМ ЕЁ ОТЛИЧИЕ JPA ОТ JDO?**

JPA (Java Persistence API) и Java Data Objects (JDO) две спецификации сохранения java объектов в базах данных. Если JPA сконцентрирована только на реляционных базах, то JDO более общая спецификация которая описывает ORM для любых возможных баз и хранилищ. Также отличаются "разработчики" спецификаций - если JPA разрабатывается как JSR, то JDO сначала разрабатывался как JSR, теперь разрабатывается как проект Apache JDO.

- **МОЖНО ЛИ ИСПОЛЬЗОВАТЬ JPA С NOSQL БАЗАМИ?**

Спецификация JPA говорит только о отображении java объектов в таблицы реляционных баз данных, но при этом существует ряд реализаций данного стандарта для noSql баз данных: Kundera, DataNucleus, ObjectDB и ряд других. Естественно, при это не все специфичные для реляционных баз данных особенности спецификации переносятся при этом на nosql базы полностью.

- **ЧТО ТАКОЕ JPQL (JAVA PERSISTENCE QUERY LANGUAGE) И ЧЕМ ОН ОТЛИЧАЕТСЯ ОТ SQL?**

JPQL (Java Persistence query language) 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 так же используются типы данных атрибутов Entity, а не полей баз данных. В отличии от SQL в JPQL есть автоматический полиморфизм. Также в JPQL используется функции которых нет в SQL: такие как KEY (ключ Map'ы), VALUE (значение Map'ы), TREAT (для приведение суперкласса к его объекту-наследнику, downcasting), ENTRY и т.п.

- **ЧТО ОЗНАЧАЕТ ПОЛИМОРФИЗМ (POLYMORPHISM) В ЗАПРОСАХ JPQL (JAVA PERSISTENCE QUERY LANGUAGE) И КАК ЕГО "ВЫКЛЮЧИТЬ"?**

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

- **ЧТО ТАКОЕ CRITERIA API И ДЛЯ ЧЕГО ОН ИСПОЛЬЗУЕТСЯ?**

Criteria API это тоже язык запросов, аналогичным JPQL (Java Persistence query language), однако запросы основаны на методах и объектах, то есть запросы выглядят так:

```
CriteriaBuilder cb = ...
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> customer = q.from(Customer.class);
q.select(customer);
```

- **ЧТО ТАКОЕ ENTITY?**

Entity это легковесный хранимый объект бизнес логики (persistent domain object). Основная программная сущность это entity класс, который так же может использовать дополнительные классы, который могут использоваться как вспомогательные классы или для сохранения состояния entity.

- **МОЖЕТ ЛИ НЕ ENTITY КЛАСС НАСЛЕДОВАТЬСЯ ОТ ENTITY КЛАССА?**

Может.

- **МОЖЕТ ЛИ ENTITY КЛАСС НАСЛЕДОВАТЬСЯ ОТ ДРУГИХ ENTITY КЛАССОВ?**

Может.

- **МОЖЕТ ЛИ ENTITY БЫТЬ АБСТРАКТНЫМ КЛАССОМ?**

Может, при этом он сохраняет все свойства Entity, за исключением того что его нельзя непосредственно инициализировать.

- **МОЖЕТ ЛИ ENTITY КЛАСС НАСЛЕДОВАТЬСЯ ОТ НЕ ENTITY КЛАССОВ (NON-ENTITY CLASSES)?**

Может.

- **КАКИЕ ТРЕБОВАНИЯ JPA К ENTITY КЛАССАМ ВЫ МОЖЕТЕ ПЕРЕЧИСЛИТЬ (НЕ МЕНЕЕ ШЕСТИ ТРЕБОВАНИЙ)?**

1. Entity класс должен быть отмечен аннотацией Entity или описан в XML файле конфигурации JPA.
2. Entity класс должен содержать public или protected конструктор без аргументов (он также может иметь конструкторы с аргументами).
3. Entity класс должен быть классом верхнего уровня (top-level class).
4. 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. Enity класс должен содержать первичный ключ, то есть атрибут или группу атрибутов которые уникально определяют запись этого Enity класса в базе данных.

- **ЧТО ТАКОЕ АТРИБУТ ENTITY КЛАССА В ТЕРМИНОЛОГИИ JPA?**

JPA указывает что она может работать как с свойствами классов (property), оформленные в стиле JavaBeans, либо с полями (field), то есть переменными класса (instance variables). Оба типа элементов Entity класса называются атрибутами Entity класса.

- **КАКИЕ ДВА ТИПА ЭЛЕМЕНТОВ ЕСТЬ У ENTITY КЛАССОВ. ИЛИ ДРУГИМИ СЛОВАМИ ПЕРЕЧИСЛИТЕ ДВА ТИПА ДОСТУПА (ACCESS) К ЭЛЕМЕНТАМ ENTITY КЛАССОВ.**

JPA указывает что она может работать как с свойствами классов (property), оформленные в стиле JavaBeans, либо с полями (field), то есть переменными класса (instance variables). Соответственно, при этом тип доступа будет либо property access или field access.

- **КАКИЕ ТИПЫ ДАННЫХ ДОПУСТИМЫ В АТРИБУТАХ ENTITY КЛАССА (ПОЛЯХ ИЛИ СВОЙСТВАХ)?**

Допустимые типы атрибутов у Entity классов:

1. примитивные типы и их обертки Java,
2. строки,
3. любые сериализуемые типы Java (реализующие Serializable интерфейс),
4. enums;
5. entity types;
6. embeddable классы
7. и коллекции типов 1–6

- **КАКИЕ ТИПЫ ДАННЫХ МОЖНО ИСПОЛЬЗОВАТЬ В АТРИБУТАХ, ВХОДЯЩИХ В ПЕРВИЧНЫЙ КЛЮЧ ENTITY КЛАССА (СОСТАВНОЙ ИЛИ ПРОСТОЙ), ЧТОБЫ ПОЛУЧЕННЫЙ ПЕРВИЧНЫЙ КЛЮЧ МОГ ИСПОЛЬЗОВАТЬСЯ ДЛЯ ЛЮБОЙ БАЗЫ ДАННЫХ? А В СЛУЧАЕ АВТОГЕНЕРИРУЕМОГО ПЕРВИЧНОГО КЛЮЧА (GENERATED PRIMARY KEYS)?**

Допустимые типы атрибутов, входящих в первичный ключ:

1. примитивные типы и их обертки Java,
2. строки,
3. BigDecimal и BigInteger,
4. java.util.Date и java.sql.Date,В случае автогенерируемого первичного ключа (generated primary keys) допустимы только числовые типы,В случае использования других типов данных в первичном ключе, он может работать только для некоторых баз данных, т.е. становится не переносимым (not portable).

- **ЧТО ТАКОЕ ВСТРАИВАЕМЫЙ (EMBEDDABLE) КЛАСС?**

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

- **МОЖЕТ ЛИ ВСТРАИВАЕМЫЙ (EMBEDDABLE) КЛАСС СОДЕРЖАТЬ ДРУГОЙ ВСТРАИВАЕМЫЙ (EMBEDDABLE) КЛАСС?**

Да, может.

- **МОЖЕТ ЛИ ВСТРАИВАЕМЫЙ (EMBEDDABLE) КЛАСС СОДЕРЖАТЬ СВЯЗИ (RELATIONSHIP) С ДРУГИМИ ENTITY ИЛИ КОЛЛЕКЦИЯМИ ENTITY? ЕСЛИ МОЖЕТ, ТО СУЩЕСТВУЮТ ЛИ КАКИЕ-ТО ОГРАНИЧЕНИЕ НА ТАКИЕ СВЯЗИ (RELATIONSHIP)?**

Может, но только в случае если такой класс не используется как первичный ключ или ключ map'ы.

- **КАКИЕ ТРЕБОВАНИЯ JPA УСТАНАВЛИВАЕТ К ВСТРАИВАЕМЫМ (EMBEDDABLE) КЛАССАМ?**

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 отмечен аннотацией Embeddable или описан в XML файле конфигурации JPA.

- **КАКИЕ ТИПЫ СВЯЗЕЙ (RELATIONSHIP) МЕЖДУ ENTITY ВЫ ЗНАЕТЕ (ПЕРЕЧИСЛИТЕ ВОСЕМЬ ТИПОВ, ЛИБО УКАЖИТЕ ЧЕТЫРЕ ТИПА СВЯЗЕЙ, КАЖДУЮ ИЗ КОТОРЫХ МОЖНО РАЗДЕЛИТЬ ЕЩЁ НА ДВА ВИДА)?**

Существуют следующие четыре типа связей

- OneToOne (связь один к одному, то есть один объект Entity может связан не больше чем с один объектом другого Entity),
- OneToMany (связь один ко многим, один объект Entity может быть связан с целой коллекцией других Entity),
- ManyToOne (связь многие к одному, обратная связь для OneToMany),
- ManyToMany (связь многие ко многим).

Каждую из которых можно разделить ещё на два вида:

- 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.

• ЧТО ТАКОЕ MAPPED SUPERCLASS?

Mapped Superclass это класс от которого наследуются Entity, он может содержать аннотации JPA, однако сам такой класс не является Entity, ему не обязательно выполнять все требования установленные для Entity (например, он может не содержать первичного ключа). Такой класс не может использоваться в операциях EntityManager или Query. Такой класс должен быть отмечен аннотацией MappedSuperclass или соответственно описан в xml файле.

• КАКИЕ ДВА ТИПА FETCH СТРАТЕГИИ В JPA ВЫ ЗНАЕТЕ?

В JPA описаны два типа fetch стратегии:

- LAZY – данные поля будут загружены только во время первого доступа к этому полю,
- EAGER – данные поля будут загружены немедленно.

• КАКИЕ ТРИ ТИПЫ СТРАТЕГИИ НАСЛЕДОВАНИЯ МАПИНГА (INHERITANCE MAPPING STRATEGIES) ОПИСАНЫ В JPA?

В JPA описаны три стратегии наследования мапинга (Inheritance Mapping Strategies), то есть как JPA будет работать с классами-наследниками Entity:

- одна таблица на всю иерархию наследования (a single table per class hierarchy) – все enity, 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.

- объединяющая стратегия (joined subclass strategy) – в этой стратегии каждый класс enity сохраняет данные в свою таблицу, но только уникальные колонки (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.
- одна таблица для каждого класса (table per concrete class strategy) – тут все просто каждый отдельный класс-наследник имеет свою таблицу, т.е. для cats и dogs 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.