Java ORM

BOOKS

* EJB в действии (Панда) (388)

VIDEO

Основы

* JPA является прослойкой между POJO и JDBC
* ORM должно для каждой структуры БД должен поддерживать возможность отображения в произвольную предметную область, это проблема называется несоответствием интерфейсов (различия между объектно-ориентируемой и реляционной парадигмой)
* Предметная модель - концептуальное представление задачи, решаемой приложением. Манипуляция объектами – это прикладная логика

JPA

* @Entity – превращает класс в сущность JPA
* Обязательно должен быть public или protected конструктор по умолчанию
* Сущности можно наследовать, при этом родитель не обязательно может быть помечен как Entity, при этом все поля родителя будут сохранены (если помечен) или отброшены если нет
* Сущности могут быть абстрактными, это допустимо, но они не могут использоваться для создания экземпляра
* Сущность может храниться в нескольких таблицах
* @Table(name="Tabname") – обозначает таблицу в которой хранятся данные, так же в этой аннотации можно определить схему (schema=) но это является плохой практикой, т.к. ее лучше указывать в настройках
* Источник данных по умолчанию определен в java:comp/DefaultDataSource, внедрить его можно с помощью @Resource
* Для отображения сущности в несколько таблиц используем @SecondaryTable (для одной) или @SecondaryTables (для нескольких). Общий вид: @SecondaryTable(name=”secondTable”, pkJoinColumns=@PrimaryKeyJoinColumn(name=”USER\_ID”)) – где secondTable.USER\_ID связан с первичным ключом таблицы Tabname
* По умолчанию все свойства класса имеющие public или protected методы соответствующие соглашению get/set отображает в одноименные столбцы
* Если нам нужно отобразить в столбцы с другим наименованием, то применяем аннотацию @Column, стоит обратить внимание на атрибуты insertable и updatetable – если их выставить в false то JPA не будет включать их в insert/update
* Если @Column разместить перед переменными экземпляра, то такой способ называется доступ к данным на основе полей (в таком случае аннотацию можно разместить и перед приватным полем и при этом запись и чтение будет непосредственно в переменную), если @Column разместить перед методом чтения (перед записи проигнорируется) то чтение и запись будет в помощью методов, это называется доступ на основе свойств
* Доступ к полям можно указать и конкретно аннотацией @Access, сначала на уровне класса (после @Entity) потом после @Column. Указывается один из двух видов доступа AccessType.FIELD или AccessType.PROPERTY
* Если у класса доступ на основе полей и отдельно мы хотим указать у поля на основе свойств, то у самого поля нужно указать @Transient чтобы предотвратить попытку отобразить поле дважды
* По сути @Transient говорит что это поле не нужно сохранять
* В БД время хранится в DATE, TIME и TIMESTAMP и аннотация @Temporal позволяет указать в каком формате время хранить (TemporalType.Date и т.д.). JPA может отображать даты в util.Date и util.Calendar или java.sql.Date, Time, Timestamp
* Перечисления: отсутствуют в БД, но с этим поможет @Enumerated (EnumType.STRING – строковое представление, EnumType.ORDINAL - индекс), соответственно это и будет отображено в БД ( по умолчанию хранится индекс)
* Коллекции: @ElementCollection, так же необходимо указать следом @CollectionTable (name = "BLA")
* Для того чтобы сделать объекты уникальными как в БД существует три способа: @Id, @IdClass, @EmbeddedId
* @Id – если первичный ключ таблицы состоит из единственного столбца, тогда просто помечаем поле этой аннотацией. Применимо к простым типам или к типам поддерживающим интерфейс Serializable
* @IdClass – если первичный ключ таблицы состоит из нескольких столбцов. Для начала определяем каждый из составного ключа аннотацией @Id, далее определяем новый класс состоящий только из ключей, который реализовывает Serializable. Так же в этом классе должен быть корректно реализован метод equals. Потом перед определением класса, который изначально используем, вставляем @IdClass (ClassKey.class)
* @EmbeddedId – второй способ отображения ключа в несколько полей. Так же создаем класс для первичного ключа, помечаем его как @Embedable (Serializable теперь не обязателен). В основном классе два поля из ключа убирают и заменяют на объект класса ключей (private ClassId classid;) и помечаем его как @EmbeddedId. Эти способом можно внедрять не только первичные ключи, но и любые другие составные поля
* Генерация первичных ключей: с помощью аннотации GeneratedValue (@GeneratedValue (strategy=GenerationType.AUTO). Есть несколько стратегий генерации: AUTO – используется по умолчанию и отдает генерацию ключа БД, IDENTITY – используется столбец специального типа, поддерживается не всеми БД, SEQUENCE – если для создания используется последовательность (в этом случае дополнительно описываем последовательность @SequenceGenerator(name=" SEQGEN", sequenceName="USER\_SEQ", initialValue=1, allocationSize=10) и далее указываем в GeneratedValue generator="SEQGEN"), TABLE – если используется таблица с номерами последовательностей (дополнительно описываем последовательность @TableGenerator(name="USER\_TABGEN", table="SEQ\_TAB", pkColumnName = "SEQ\_NAME", valueColumnName = "SEQ\_VAL", pkColumnValue="SEQ1") и далее указываем generator в GeneratedValue)
* Так же первичный ключ можно сгенерировать программно – помечаем метод аннотацией @PrePersist и он будет вызываться перед вставкой (  
  @PrePersist  
  public void genKey(){ userId = UUID.randomUUID().toString(); }
* Отношения между сущностями :  
  - один к одному: помечается @OneToOne (само поле protected Info info; и сущность @Entity class Info), так же у этой аннотации есть свойства targetEntity – класс объекта сущности (на случай когда отношения определяются интерфейсами), cascade – как распространяются данные, fetch – как извлекаются данные, optional – интерпретирует отношение как необязательное (если false, то этот экземпляр не может существовать без ссылаемого) . В общем случае это отношение однонаправленное, чтобы сделать его двунаправленным нужно дополнительно в ссылаемом классе указать @OneToOne (mappedBy="info", optional=false), указываем в качестве mappedBy именно поле  
  - один ко многим: помечается @ManyToOne у одного класса (Info) для поля protected User user; и @OneToMany(mappedBy="user") у другого (User) для поля protected Set<Info> infos;  
  - многие ко многим: отмечаем аннотацией @ManyToMany поля типа Set
* Свойство mappedBy по сути определяет владельца связи, определяется в подчиненной сущности
* В java объекты можно наследовать, в БД нет, поэтому эту проблему решает отображение наследования. Существует три стратегии: единой таблицы, соединения таблиц и отдельных таблиц для каждого класса
* Стратегия единой таблицы (по умолчанию) : все классы в иерархии наследования отображаются в единую таблицу. Поэтому таблица будет содержать поля присущие каждому классу. Порядок такой: 1) для класса родителя задаем @Inheritance(strategy=InheritanceType.SINGLE\_TABLE) и @DescriminatorColumn(name="USER\_TYPE", descriminatorType=DescriminatorType.STRING, length=1) 2) а у классов наследников указываем @DecriminatorValue(value="S"). Дескриминатор служит для того чтобы различать конечные классы
* Стратегия соединения таблиц: для каждой сущности и для родителя создается своя таблица и они соединяются как один к одному. 1) для родителя задаем @Inheritance(strategy=InheritanceType.JOINED) и @DescriminatorColumn(name="USER\_TYPE", descriminatorType=DescriminatorType.STRING, length=1) 2) у наследников задаем @DecriminatorValue(value="S") и @PrimaryKeyJoinColumn(name="USER\_ID"). Т.е. здесь мы указываем по какому столбцу будут соединяться таблицы. С точки зрения архитектуры БД это лучший выбор, но менее производительный, т.к. нужно соединять таблицы
* Стратегия отдельных таблиц: так же создается отдельная таблица, но между ними отсутствуют отношения, все данные из разделяемой таблицы не разделяются, а дублируются. 1) указываем у родительского класса @Inheritance(strategy=InheritanceType.TABLE\_PER\_CLASS)
* Управляет жизненным циклом сущностей объект EntityManager
* Жизненный цикл сущностей: EntityManager ничего не знает о сущности пока мы не подключим ее, когда мы подключаем ее то гарантируется загрузка данных из БД (подключить сущность можно с помощью persist или merge, а так же при выполнении find и query, обновить сущность – refresh)
* Существует два типа областей видимости контекста сохранения: область видимости транзакции (ограничена транзакцией, после завершения сущности автоматически отключатся) и расширенная область видимости (распространяет свое действие на несколько транзакций, однако может работать только с сеансовыми компонентами с сохранением состояния и действует пока существует компонент)
* Внедрить EntityManager в EJB: @PersistenceContext private EntityManager entityManager; (опционально используется атрибут unitName, который указывает на имя единицы хранения (группы сущностей) они описаны в persistence.xml, например, general и admin), атрибут type определяет область видимости для EntityManager (PersistenceContexType.EXTENDED или .TRANSACTION)
* EntityManager не поддерживает работу в многопоточной среде и лучше его использовать внутри компонента EJB
* Если не внедрять EntityManager с помощью контейнера, то есть способ внедрить его с помощью EntityManagerFactory (entityManagerFactory.createEntityManager()) но полученный таким образом EntityManager не участвует в транзакции, поэтому его нужно подключить вручную (entityManager.joinTransaction()) и перед этим необходимо указать аннотацию @PersistenceUnit
* EntityManager гарантирует автоматическое сохранение изменений в БД для подключенных сущностей, это наиболее привлекательное из ORM т.к. от нас скрывается синхронизация. Обычно запись происходит во время завершения транзакции, но есть возможность сделать это и принудительно с помощью flush, причем flush коснется всех подключенных сущностей
* Поиск значений по ключу - существует несколько способов : 1) entityManager.find(Seller.class, sellerId), если сущность не найдена вернется null, если выполнить его вне контекста транзакций, то вернется неподключенная сущность. Важная особенность – метод find использует внутренний кэш EntityManager
* По умолчанию при извлечении сущности извлекается полностью вся информация (жадное извлечение), но так же есть возможность сделать отложенное извлечение. Есть несколько способов: 1) отметить поле как @Basic(fetch=FetchType.LAZY), на самом деле для связанных сущностей тоже можно указать атрибут fetch
* Удалить сущность можно как EntityManager.remove()
* Запрос сущностей: find ищет данные по первичному ключу, если мы хотим искать по другим полям то используем интерфейс javax.persistence.Query (на я зыке JPQL), javax.persistence.criteria.CriteriaQuery (запрос сконструированный с помощью метамодели)
* Можно создавать динамические запросы: entityManager.createQuery("select \* from ..", Category.class); query.getResultList(); или именованные запросы : entityManager.createNamedQuery("findAllCategories", Category.class);