**Criteria API , JPQL или HQL**

Criteria API лучше подходит для динамически генерируемых запросов, когда фильтры запросов динамически применяются во время выполнения. Поэтому для предотвращения атак SQL-инъекций при построении динамических запросов Criteria API является очень хорошим выбором.

Критерии запросов менее выразительны, и вы можете легко получить очень сложный и неэффективный запрос, сгенерированный SQL. Однажды я присоединился к крупному корпоративному приложению, в котором Criteria API был методом запросов по умолчанию, и даже тщательный анализ кода не смог преодолеть ужас незнания того, с какими SQL-запросами мы собираемся работать.

JPQL или HQL гораздо более выразительны, и намного проще предсказать связанный сгенерированный запрос SQL. Также гораздо проще просматривать свои HQL-запросы, чем критерии.

В большинстве случаев использования запросов сущностей не требуются динамические предложения where, поэтому вы можете реализовать большинство запросов с JPQL, оставляя критерии для динамических.

Стоит отметить, что выбор сущностей с помощью JPQL или Criteria API имеет смысл, если вам необходимо изменить их. В противном случае проекция DTO(*загрузка данных во временные объекты*) работает лучше.

Стратегии:

**1. Lazy/ Eager**

**2. Именованный граф/ Динамический граф**

**3. Select / Join Fetch**

**4. JPQL / JPA Criteria/ DTO in SQL**(загрузка данных во временные объекты)

Как выбрать стратегию?

Резюмируя, я хочу обсудить преимущества и недостатки представленных стратегий и гайд для выбора подходящей стратегии.

**Lazy Loading**

Преимущества:

* Вы выгружаете только те объекты, которые прямо запросили
* Вы не должны думать о конкретной стратегии выгрузки
* Ваш корневой узел загружается быстро

Недостатки:

* Можно получить большие задержки при выгрузки объектов, которые имеют длинный путь в графе
* Нельзя использовать джойны SQL для загрузки частей графа, Вы производите несколько выражений select
* Вы не гибки, потому что Ваша стратегия задана глобально на уровне маппинга

Нужно использовать если:

* Ваш фронтенд работает на той же самой JVM, что и бэкенд с Hibernate
* Вы не можете предвидеть какие из частей графа нужны в каждой ситуации
* Вы хотите, чтобы приложение стартовало быстро и распределяете нагрузку во времени между последующими действиями пользователя

Вы не должны использовать если:

* Ваш фронтенд вызывает бэкенд Hibernate удаленно
* очевидно какие именно части графа должны быть загружены
* Вы можете пренебречь временем запуска приложения и предзагрузить большинство данных

**Eager Loading**

Преимущества:

* Вы явно загружаете все данные, которые необходимы
* Вы можете использовать JPA для оптимальной стратегии выгрузки (батчи, джойны, селекты)
* Вы не имеете дело с закрытыми сессиями

Недостатки:

* Вы должны думать, какие части должны быть заранее выгружены и какие нет
* Вы будете иметь высокую задержку для загрузки корневого объекта из графа, потому что также другие части графа должны быть загружены
* Вероятно Вы выгрузите ненужные части графа
* Вы не гибки из-за того что стратегия глобальна и задается на уровне маппинга

Нужно использовать если:

* ясно какие объекты из графа всегда выгружаются рядом
* Вам нужно сделать предзагрузку для быстрого доступа позже

Вы не должны использовать если:

* не ясно какие части графа требуются выгружать
* граф выгрузки очень большой

**Join Fetch / Join Fetch с «distinct» – все в одном запросе (т.к. стратегия Select*(все в разных запросах)* идет по умолчанию)**

Преимущества:

* Вы решаете на уровне операций что нужно выгружать, без выгрузки ненужных объектов
* Вы можете предложить разные варианты Вашей операции (например, и lazy и eager одновременно) используя разные стратегии join для одной и той же операции

Недостатки:

* Более сложно определить что выгружать на уровне запроса
* Опреление что нужно загружать на уровне запроса делает запрос менее переиспользуемым

Нужно использовать если:

* для определенной операции, всегда одни и теже данные должны быть выгружены

Вы не должны использовать если:

* Методы с джойном немного менее эффективны, потому что Вы получаете декартово произведение

**Именованные графы *(при задании графа используются аннотации)***

Преимущества:

* Вы решаете на уровне операции что нужно выгрузить
* Вы можете предложить разные варианты операции, позволяя использовать разные графы загрузки
* Увеличивается переиспользование, потому что Вы используете тот же самый граф для разных операций

Недостатки:

* Синтаксис многословный, если граф становится сложней
* Именованные графы не могут быть совмещены

Нужно использовать если:

* у Вас есть несколько **стратегий** загрузки *(****т.е. комбинаций определяющих, какие из дочерних таблиц должны быть загружены, а какие нет****)*, которые должны поддерживаться и могут переиспользованы для разных операций в репозитории/DAO

Вы не должны использовать если:

* Есть очень много **стратегий** или комбинаций стратегий

**Динамические графы *(условие прописывается в коде непосредственно перед запросом)***

Преимущества:

* Вы решаете на уровне операции что нужно выгрузить
* Потребитель операции репозитория/DAO может сам решить что выгружать, это может уменьшить количество операций предлагаемых Вашими репозиториями/DAO

Недостатки:

* Сложно задавать
* Сложно переиспользовать, потому что они действительны только для конкретного случая

Нужно использовать если:

* есть множество разных **стратегий** загрузки или комбинаций *(****т.е. комбинаций определяющих, какие из дочерних таблиц должны быть загружены, а какие нет****)*

Вы не должны использовать если:

* Вам нужно малое количество **стратегий** в Вашем DAO

**Criteria API**

Преимущества:

* Позволяет писать полностью динамические запросы, включая динамические стратегии
* Может быть комбинирован с графами

Недостатки:

* Более сложен, труден для чтения и сложно задавать

Нужно использовать если:

* у Вас есть множество вариантов одного и того же **запроса**
* Вы используете конкатенации String для создания **запросов**

Вы не должны использовать если:

* Вы можете сделать то же самое одним статическим **запросом**

**DTO**

Когда выбранная сущность в процессе работы изменяться не будет работает лучше DTO (загрузка данных во временные объекты) чем JPQL или Criteria API в случае.