Red Hat JBoss Enterprise Application Platform 7.4

Hibernate 애플리케이션 개발

Jakarta Persistence API(JPA) 또는 Hibernate 애플리케이션을 Red Hat JBoss Enterprise Application Platform용 개발 및 배포하고자 하는 개발자 및 관리자를 위한 지침 및 정보.

Last Updated: 2024-02-10

Red Hat JBoss Enterprise Application Platform 7.4 Hibernate 애플리케이션 개발

Jakarta Persistence API(JPA) 또는 Hibernate 애플리케이션을 Red Hat JBoss Enterprise Application Platform용 개발 및 배포하고자 하는 개발자 및 관리자를 위한 지침 및 정보.

법적 공지

초록

이 문서에서는 Red Hat JBoss Enterprise Application Platform을 사용하여 Jakarta Persistence 또는 Hibernate 애플리케이션을 개발 및 배포하려는 개발자와 관리자를 위한 정보를 제공합니다.

차례

JBOSS EAP 문서에 대한 피드백 제공

오류를 보고하거나 문서를 개선하기 위해 Red Hat Jira 계정에 로그인하여 문제를 제출하십시오. Red Hat Jira 계정이 없는 경우 계정을 생성하라는 메시지가 표시됩니다.

절차

1. **티켓을 생성하려면** 다음 링크를 클릭하십시오.
2. **문서 URL**, **섹션 번호** 를 포함하고 **문제를 설명하십시오**.
3. **요약** 에 문제에 대한 간략한 설명을 입력합니다.
4. 설명에서 문제 또는 개선 사항에 대한 자세한 설명을 제공합니다. **문서에서 문제가 발생한 위치에 URL을 포함합니다.**
5. **Submit 을 클릭하고 문제를 적절한 문서 팀으로 라우팅합니다.**

보다 포괄적 수용을 위한 오픈 소스 용어 교체

Red Hat은 코드, 문서, 웹 속성에서 문제가 있는 용어를 교체하기 위해 최선을 다하고 있습니다. 먼저 마스터(master), 슬레이브(slave), 블랙리스트(blacklist), 화이트리스트(whitelist) 등 네 가지 용어를 교체하고 있습니다. 이러한 변경 작업은 작업 범위가 크므로 향후 여러 릴리스에 걸쳐 점차 구현할 예정입니다. 자세한 내용은 CTO Chris Wright의 메시지를 참조하십시오.

1장. 소개

1.1. HIBERNATE CORE 정보

Hibernate Core는 Java 언어용 객체 관계 매핑 프레임워크입니다. 개체 지향 도메인 모델을 관계형 데이터베이스에 매핑하기 위한 프레임워크를 제공하므로 애플리케이션이 데이터베이스와의 직접적인 상호 작용을 방지할 수 있습니다. Hibernate는 영구적인 데이터베이스 액세스를 고수준 오브젝트 처리 기능으로 교체하여 객체 관계상의 임피던스 불일치 문제를 해결합니다.

1.2. HIBERNATE ENTITYMANAGER

Hibernate EntityManager는 Jakarta Persistence 2.2 사양에 정의된 프로그래밍 인터페이스 및 라이프사이클 규칙을 구현합니다. Hibernate Annotations와 함께 이 래퍼는 성숙한 Hibernate Core 위에 독립 실행형 Jakarta Persistence 솔루션을 구현합니다. 프로젝트의 비즈니스 및 기술 요구에 따라 세 가지, 자카르타 지속성 프로그래밍 인터페이스 및 라이프사이클이 없는 주석 또는 순수 네이티브 Hibernate Core를 모두 사용할 수 있습니다. 항상 Hibernate 네이티브 API로 대체되거나 필요한 경우 네이티브 JDBC 및 SQL로 대체할 수 있습니다. JBoss EAP에 완전한 자카르타 지속성 솔루션을 제공합니다.

JBoss EAP 7.3 이상 릴리스는 Jakarta EE 8에 정의된 Jakarta Persistence 2.2 사양과호환됩니다.

Hibernate는 사양에 추가 기능도 제공합니다. Jakarta Persistence 및 JBoss EAP를 시작하려면 JBoss EAP와 함께 제공되는 **bean-validation,greeter** 및 **equipmentsink** 빠른 시작을 참조하십시오.

Jakarta Persistence는 Jakarta Enterprise Beans 3 또는 최신 Jakarta Contexts and Dependency Injection과 같은 컨테이너에서 사용할 수 있으며 특정 컨테이너 외부에서 실행되는 독립 실행형 Java SE 애플리케이션에서 사용할 수 있습니다. 다음 프로그래밍 인터페이스와 아티팩트는 두 환경 모두에서 사용할 수 있습니다.

중요

Hibernate와 함께 보안 관리자를 사용하려면 **EntityManagerFactory** 가 JBoss EAP 서버에서 부트스트랩된 경우에만 Hibernate가 이를 지원합니다. 애플리케이션에서 **EntityManagerFactory** 또는 **SessionFactory** 를 부트스트랩하는 경우 지원되지 않습니다.

EntityManagerFactory

: 엔터티 관리자 팩토리는 엔터티 관리자 인스턴스를 제공하고, 모든 인스턴스가 동일한 데이터베이스에 연결하도록 구성되고, 특정 구현에서 정의한 것과 동일한 기본 설정을 사용하도록 구성됩니다. 여러 엔터티 관리자 팩토리를 준비하여 여러 데이터 저장소에 액세스할 수 있습니다. 이 인터페이스는 기본 Hibernate의 SessionFactory와 유사합니다.

EntityManager

: EntityManager API는 특정 작업 단위의 데이터베이스에 액세스하는 데 사용됩니다. 영구 엔터티 인스턴스를 생성 및 제거하고, 기본 키 ID로 엔터티를 찾고, 모든 엔터티를 쿼리하는 데 사용됩니다. 이 인터페이스는 Hibernate의 세션과 유사합니다.

지속성 컨텍스트

: 지속성 컨텍스트는 영구 엔터티 ID에 대해 고유한 엔터티 인스턴스가 있는 엔터티 인스턴스 집합입니다. 지속성 컨텍스트 내에서 엔터티 인스턴스 및 해당 라이프사이클은 특정 엔터티 관리자가 관리합니다. 이 컨텍스트의 범위는 트랜잭션 또는 확장된 작업 단위일 수 있습니다.

지속성 단위

지정된 엔터티 관리자가 관리할 수 있는 엔터티 유형 집합은 지속성 유닛에서 정의합니다. 지속성 유닛은 애플리케이션에서 관련 또는 그룹화된 모든 클래스 집합을 정의하고 단일 데이터 저장소에 매핑에 배치해야 합니다.

컨테이너 관리 엔터티 관리자

컨테이너에서 라이프사이클을 관리하는 엔터티 관리자입니다.

애플리케이션 관리 엔터티 관리자

애플리케이션에서 라이프사이클을 관리하는 엔터티 관리자.

자카르타 트랜잭션 엔터티 관리자

자카르타 트랜잭션과 관련된 엔터티 관리자.

resource-local 엔터티 관리자

리소스 트랜잭션을 사용하는 엔터티 관리자(Norizon Transactions 트랜잭션이 아님).

추가 리소스

- 빠른 시작을 다운로드하고 실행하는 방법에 대한 자세한 내용은 JBoss EAP *Getting Started Guide 의 빠른 시작 예제 사용을 참조하십시오.*
- 보안 관리자에게 대한 자세한 내용은 JBoss EAP의 Java Security Manager 를 참조하십시오. *서버 보안 구성 방법*.

2장. HIBERNATE 구성

2.1. HIBERNATE 구성

애플리케이션 서버 내부와 독립 실행형 애플리케이션의 엔터티 관리자에 대한 구성은 지속성 아카이브에 있습니다. 지속성 아카이브는 **META-INF/** 폴더에 있는**persistence.xml** 파일을 정의해야 하는 JAR 파일입니다.

persistence.xml 파일을 사용하여 데이터베이스에 연결할 수 있습니다. 이 작업을 수행하는 방법에는 두 가지가 있습니다.

- JBoss EAP의 **datasources** 하위 시스템에서 구성된 데이터 소스 지정.
  jta-data-source 는 이 지속성 유닛이 매핑하는 데이터 소스의 Java Naming 및 Directory Interface 이름을 가리킵니다. 여기에서 **java:jboss/datasources/ExampleDS** 는 JBoss EAP에 포함된 **H2 DB** 를 가리킵니다.

  persistence.xml 파일의 **object-relational-mapping** 예

  ```
  <persistence>
    <persistence-unit name="myapp">
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
      <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
      <properties>
        ... ...
      </properties>
    </persistence-unit>
  </persistence>
  ```

- 연결 속성을 지정하여 명시적으로 **persistence.xml** 파일 구성.

  persistence.xml 파일에서 연결 속성을 지정하는 예

  ```
  <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver"/>
  <property name="javax.persistence.jdbc.user" value="sa"/>
  <property name="javax.persistence.jdbc.password" value=""/>
  <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:."/>
  ```

  연결 속성의 전체 목록은 **persistence.xml** 파일에서 Connection Properties Configurable을 참조하십시오.

런타임 시 Hibernate의 동작을 제어하는 다양한 속성이 있습니다. 모두 선택 사항이며 적절한 기본값이 있습니다. 이러한 Hibernate 속성은 모두 **persistence.xml** 파일에서 사용됩니다. 구성 가능한 모든 Hibernate 속성의 전체 목록은 Hibernate Properties 를 참조하십시오.

2.2. 2차 캐시

2.2.1. 두 번째 수준 캐시 정보

두 번째 수준 캐시는 애플리케이션 세션 외부에서 지속되는 정보를 보관하는 로컬 데이터 저장소입니다. 캐시는 애플리케이션과 별도로 데이터를 유지하여 런타임을 개선하여 지속성 프로바이더가 관리합니다.

JBoss EAP는 다음과 같은 목적으로 캐싱을 지원합니다.

- 웹 세션 클러스터링
- 상태 저장 세션 빈 클러스터링
- SSO 클러스터링
- Hibernate 두 번째 수준 캐시
- 자카르타 지속성 두 번째 수준 캐시

주의

각 캐시 컨테이너는 **repl** 및 **dist** 캐시를 정의합니다. 이러한 캐시는 사용자 애플리케이션에서 직접 사용해서는 안 됩니다.

2.2.2. Hibernate에 대해 두 번째 레벨 캐시 구성

Hibernate의 두 번째 수준 캐시 역할을 하는 Infinispan의 구성은 다음 두 가지 방법으로 수행할 수 있습니다.

- JBoss EAP *개발 가이드에* 설명된 대로 **persistence.xml** 파일을 사용하여 Jakarta Persistence 애플리케이션을 통해 두 번째 수준 캐시를 구성하는 것이 좋습니다.
- 또는 아래에 설명된 대로 **hibernate.cfg.xml** 파일을 사용하여 Hibernate 네이티브 애플리케이션을 통해 두 번째 수준 캐시를 구성할 수 있습니다.

Hibernate 네이티브 애플리케이션을 사용하여 Hibernate용 두 번째 레벨 캐시 구성

1. 배포의 클래스 경로에 **hibernate.cfg.xml** 파일을 생성합니다.
2. 다음 XML을 **hibernate.cfg.xml** 파일에 추가합니다. XML은 **<session-factory>** 태그 내에 있어야 합니다.

```
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
<property
name="hibernate.cache.region.factory_class">org.jboss.as.jpa.hibernate5.infinispan.InfinispanRegionFactory</property>
```

3. 애플리케이션 내에서 Hibernate 네이티브 API를 사용하려면 **MANIFEST.MF** 파일에 다음 종속성을 추가해야 합니다.

```
Dependencies: org.infinispan,org.hibernate
```

3장. HIBERNATE 주석

3.1. HIBERNATE 주석

org.hibernate.annotations 패키지에는 표준 Jakarta Persistence 주석 위에 Hibernate에서 제공하는 몇 가지 주석이 포함되어 있습니다.

표 3.1. 일반 주석

주석	설명
검사	클래스, 속성 또는 수집 수준에서 정의할 수 있는 임의의 SQL 검사 제약 조건입니다.
변경할 수 없음	엔터티 또는 컬렉션을 변경할 수 없음으로 표시합니다. 주석이 없음은 요소가 변경됨을 의미합니다. 애플리케이션에서 변경할 수 없는 엔터티를 업데이트할 수 없습니다. 변경 불가능한 엔터티에 대한 업데이트는 무시되지만 예외는 발생하지 않습니다. **@immutable** 을 컬렉션에 배치하면 컬렉션은 변경할 수 없으므로 컬렉션에 추가 및 삭제가 허용되지 않습니다. 이 경우 **HibernateException** 이 발생합니다.

표 3.2. 캐싱 엔티티

주석	설명
캐시	캐싱 전략을 루트 엔터티 또는 컬렉션에 추가합니다.

표 3.3. 수집 관련 주석

주석	설명
MapKeyType	영구 맵의 키 유형을 정의합니다.
ManyToAny	다양한 엔터티 유형을 가리키는 **ToMany** 연결을 정의합니다. 엔터티 유형 일치는 메타데이터 차별자 열을 통해 수행됩니다. 이러한 유형의 매핑은 경계만 있어야 합니다.
OrderBy	SQL 순서를 사용하여 컬렉션 주문(HQL 순서 아님).
OnDelete	컬렉션, 배열 및 결합된 하위 클래스 삭제에 사용할 전략입니다. 현재 보조 테이블의 **OnDelete** 는 지원되지 않습니다.
Persister	사용자 지정 지속자를 지정합니다.

주석	설명
sort	수집 정렬(Java 레벨 정렬).
다음과 같습니다.	컬렉션의 요소 Entity 또는 target 엔터티에 추가할 절입니다. 절은 SQL로 작성됩니다.
WhereJoinTable	collection 조인 테이블에 추가할 절이 있습니다. 절은 SQL로 작성됩니다.

표 3.4. CRUD 작업을 위한 사용자 지정 SQL

주석	설명
로더	Hibernate 기본 **FIND 메서드를** 덮어씁니다.
SQLDelete	Hibernate 기본 **DELETE** 메서드를 덮어씁니다.
SQLDeleteAll	Hibernate 기본 **DELETE ALL** 메서드를 덮어씁니다.
SQLInsert	Hibernate 기본 **INSERT INTO** 메서드를 덮어씁니다.
SQLUpdate	Hibernate 기본 **UPDATE** 방법을 덮어씁니다.
하위 선택	변경 불가능한 읽기 전용 엔터티를 지정된 SQL 하위 선택 표현식에 매핑합니다.
동기화	자동 플러시가 올바르게 수행되고 파생된 엔터티에 대한 쿼리가 오래된 데이터를 반환하지 않는지 확인합니다. **Subselect** 와 함께 주로 사용됩니다.

표 3.5. 엔터티

주석	설명
cascade	연관에 캐스캐드 전략 적용.

주석	설명
엔터티	표준 **@Entity** 에 정의된 것 외에도 필요할 수 있는 추가 메타데이터를 추가합니다. ● **mutable**: 이 엔티티가 변경 가능한지 여부 ● **dynamicInsert**: 동적 SQL을 삽입 허용 ● **dynamicUpdate**: 동적 SQL 업데이트 허용 ● **selectBeforeUpdate**: 객체가 실제로 수정되지 않는 한 Hibernate가 SQL UPDATE를 수행하지 않도록 지정합니다. ● Polymorphism **:** 엔터티 다형성이 PolymorphismType.IMPLICIT (기본값) 또는 PolymorphismType.EXPLICIT인지 여부 ● **OpimisticLock**: 최적의 잠금 전략 (OptimisticLockType.VERSION, OptimisticLockType.NONE, OptimisticLockType.DIRTY 또는 OptimisticLockType.ALL)  **참고** 주석 "Entity"는 더 이상 사용되지 않으며 향후 릴리스에서 제거될 예정입니다. 개별 속성 또는 값은 주석이 되어야 합니다.
다형성	다형성 Hibernate의 유형을 정의하는 데 사용되는 Hibernate는 엔터티 계층 구조에 적용됩니다.
proxy	특정 클래스의 지연 및 프록시 구성.
테이블	1차 또는 보조 테이블에 대한 보완 정보.
테이블	테이블의 복수 주석.
대상	명시적 타겟을 정의하여 반영 및 일반 해결을 방지합니다.
Tuplizer	엔터티 또는 구성 요소에 대한 매개 변수를 정의합니다.
Tuplizers	엔터티 또는 구성 요소에 대한 매개 변수 집합을 정의합니다.

표 3.6. 가져오기 중

주석	설명
BatchSize	SQL 로드를 위한 배치 크기.
FetchProfile	가져오기 전략 프로필을 정의합니다.
FetchProfiles	**@FetchProfile** 에 대한 복수형 주석.
LazyGroup	엔터티 특성을 동일한 그룹에 속하는 다른 모든 속성과 함께 가져와야 함을 지정합니다. 엔터티 특성을 지연해서 로드하려면 바이트코드 개선이 필요합니다. 기본적으로 수집되지 않은 속성은 모두 **DEFAULT** 라는 하나의 그룹에 로드됩니다. 이 주석을 사용하면 그룹의 하나의 속성에 액세스할 때 서로 다른 속성 그룹을 함께 정의할 수 있습니다.

표 3.7. 필터

주석	설명
필터	필터를 컬렉션의 엔터티 또는 대상 엔터티에 추가합니다.
FilterDef	필터 정의.
FilterDefs	필터 정의의 배열입니다.
FilterJoinTable	조인 테이블 컬렉션에 필터를 추가합니다.
FilterJoinTables	컬렉션에 **@FilterJoinTable** 을 여러 개 추가합니다.
필터	여러 개의 **@Filter** 추가.
ParamDef	매개 변수 정의입니다.

표 3.8. 기본 키

주석	설명
생성됨	이 주석이 있는 속성은 데이터베이스에 의해 생성됩니다.
GenericGenerator	모든 종류의 Hibernate 생성기를 detyped 방식으로 설명하는 생성기 주석.
GenericGenerators	일반 생성자 정의의 배열입니다.

주석	설명
NaturalId	속성이 엔터티의 자연 ID의 일부임을 지정합니다.
매개변수	키/값 패턴.
RowId	Hibernate의 **ROWID** 매핑 기능 지원.

표 3.9. 상속

주석	설명
디스크리미네이터-	차별자 공식을 루트 엔티티에 배치할 수 있습니다.
DiscriminatorOptions	Hibernate 특정 차별자 속성을 표현하는 선택적 주석입니다.
MetaValue	지정된 차별자 값을 해당 엔터티 유형에 매핑합니다.

표 3.10. JP-QL/HQL 쿼리 매핑

주석	설명
NamedNativeQueries	Hibernate **NamedNativeQuery 오브젝트를 보유하도록 NamedNativeQueries** 를 확장합니다.
NamedNativeQuery	Hibernate 기능을 사용하여 **NamedNativeQuery** 를 확장합니다.
NamedQueries	Hibernate **NamedQuery** 오브젝트를 유지하기 위해 **NamedQueries** 를 확장합니다.
NamedQuery	Hibernate 기능을 사용하여 **NamedQuery** 를 확장합니다.

표 3.11. 간단한 속성 매핑

주석	설명
AccessType	속성 액세스 유형.
열	열 배열을 지원합니다. 구성 요소 사용자 유형 매핑에 유용합니다.

주석	설명
ColumnTransformer	에서 값을 읽고 열에 값을 쓰는 데 사용되는 사용자 지정 SQL 표현식. 직접 객체 로딩/저장 및 쿼리에 사용합니다. 쓰기 표현식에는 값에 대한 정확히 하나의 '?' 자리 표시자가 포함되어야 합니다.
ColumnTransformers	**@ColumnTransformer** 에 대한 복수형 주석. 두 개 이상의 열에서 이 동작을 사용하는 경우 유용합니다.

표 3.12. 속성

주석	설명
공식	대부분의 위치에서 **@Column** 대신 사용합니다. 공식은 유효한 SQL 조각이어야 합니다.
인덱스	데이터베이스 인덱스를 정의합니다.
JoinFormula	대부분의 위치에서 **@JoinColumn** 의 대체로 사용됩니다. 공식은 유효한 SQL 조각이어야 합니다.
부모	속성을 소유자(일반적으로 소유 엔터티)를 포인터로 참조합니다.
유형	Hibernate 유형.
TypeDef	Hibernate 유형 정의.
TypeDefs	Hibernate 유형 정의 배열.

표 3.13. 단일 연결 관련 주석

주석	설명
Any	여러 엔터티 유형을 가리키는 **ToOne** 연결을 정의합니다. 따라 엔터티 유형 일치는 메타데이터 차별기 열을 통해 수행됩니다. 이러한 유형의 매핑은 경계만 있어야 합니다.
AnyMetaDef	**@Any** 및 **@ManyToAny** 메타데이터를 정의합니다.
AnyMetaDefs	**@Any** 및 **@ManyToAny** 메타데이터 집합을 정의합니다. 엔터티 수준 또는 패키지 수준에서 정의할 수 있습니다.
가져오기	지정된 연결에 사용되는 가져오기 전략을 정의합니다.

주석	설명
LazyCollection	컬렉션의 지연 상태를 정의합니다.
LazyToOne	ToOne 연결의 지연 상태를 정의합니다(즉, **OneToOne** 또는 **ManyToOne**).
찾을 수 없음	연관에서 요소를 찾을 수 없는 경우 수행할 작업.

표 3.14. 최적화된 잠금

주석	설명
OptimisticLock	주석이 지정된 속성을 변경하면 엔터티 버전이 증가하게 됩니다. 주석이 없으면 속성이 최적화된 잠금 전략(기본값)에 포함됩니다.
OptimisticLocking	엔터티에 적용할 최적화된 잠금 스타일을 정의하는 데 사용됩니다. 계층 구조에서 루트 엔터티에만 유효합니다.
소스	버전 및 타임스탬프 버전 속성과 함께 선택적 주석입니다. 주석 값은 타임스탬프가 생성되는 위치를 결정합니다.

4장. HIBERNATE 쿼리 언어

4.1. HIBERNATE 쿼리 언어 정보

Jakarta Persistence 쿼리 언어 소개

Jakarta Persistence 쿼리 언어는 Jakarta Persistence 사양 의 일부로 정의된 플랫폼 독립적인 개체 지향 쿼리 언어입니다.

Jakarta Persistence 쿼리 언어는 관계형 데이터베이스에 저장된 엔터티에 대한 쿼리를 만드는 데 사용됩니다. SQL에 의해 크게 영향을 미치고, 해당 쿼리는 구문의 SQL 쿼리와 유사하지만 데이터베이스 테이블에서 직접 사용하지 않고 Jakarta Persistence 엔터티 개체에 대해 작동합니다.

HQL 소개

HQL(Hibernate Query Language)은 SQL과 같이 강력한 쿼리 언어입니다. 그러나 SQL과 비교할 때 HQL은 완전히 객체 지향적이며 상속, 다형성 및 연관과 같은 개념을 이해합니다.

HQL은 Jakarta Persistence 쿼리 언어의 상위 집합입니다. HQL 쿼리는 항상 유효한 Jakarta Persistence 쿼리 언어 쿼리인 것이 아니지만 Jakarta Persistence 쿼리 언어 쿼리는 항상 유효한 HQL 쿼리입니다.

HQL 및 Jakarta Persistence 쿼리 언어는 쿼리 작업을 수행하는 비유형 보안 방법입니다. 기준 쿼리는 쿼리에 안전한 타입의 접근 방식을 제공합니다.

4.2. HQL 문 정보

HQL 및 Jakarta Persistence 쿼리 언어는 모두 **SELECT**,**UPDATE**, **DELETE** 문을 허용합니다. HQL은 **SQL INSERT -SELECT**와 유사한 형태로 **INSERT** 문을 추가로 허용합니다.

다음 표는 다양한 HQL 문에 대한 BNF(Backus-Naur Form) 표기법의 구문을 보여줍니다.

표 4.1. HQL 문

내용	설명
SELECT	HQL의 **SELECT** 문에 대한 BNF는 다음과 같습니다. select_statement :: = [select_clause] from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]

내용	설명
UPDATE	HQL의 **UPDATE** 문은 Jakarta Persistence 쿼리 언어와 동일합니다. update_statement ::= update_clause [where_clause] update_clause ::= UPDATE entity_name [[AS] identification_variable] SET update_item {, update_item}* update_item ::= [identification_variable.] {state_field | single_valued_object_field} = new_value new_value ::= scalar_expression | simple_entity_expression | NULL
DELETE	HQL의 **DELETE** 문에 대한 BNF는 자카르타 지속성 쿼리 언어와 동일합니다. delete_statement ::= delete_clause [where_clause] delete_clause ::= DELETE FROM entity_name [[AS] identification_variable]
삽입	HQL의 BNF for **INSERT** 문은 다음과 같습니다. insert_statement ::= insert_clause select_statement insert_clause ::= INSERT INTO entity_name (attribute_list) attribute_list ::= state_field[, state_field]* 여기에 해당하는 Jakarta Persistence 쿼리 언어는 없습니다.

주의

Hibernate를 사용하면 DML(Data Manipulation Language)을 사용하여 HQL(Hibernate Query Language)을 통해 매핑된 데이터베이스에서 직접 데이터를 대량 삽입, 업데이트 및 삭제할 수 있습니다.

DML을 사용하면 객체/관계 매핑을 위반할 수 있으며 개체 상태에 영향을 줄 수 있습니다. 오브젝트 상태는 메모리에 남아 있으며 DML을 사용하면 기본 데이터베이스에서 수행되는 작업에 따라 메모리 내 오브젝트의 상태에 영향을 미치지 않습니다. DML을 사용하는 경우 메모리 내 데이터를 주의해서 사용해야 합니다.

UPDATE 및 DELETE 문 정보
UPDATE 및 **DELETE** 문에 대한 의사syntax는 다음과 같습니다.

(업데이트 | 삭제)에서? EntityName (WHERE where_conditions)?.

참고

FROM 키워드 및 **WHERE** 절은 선택 사항입니다. **FROM** 절은 나머지 쿼리에 사용할 수 있는 오브젝트 모델 유형의 범위를 정의합니다. 또한 나머지 쿼리에서 사용할 수 있는 모든 식별 변수를 정의합니다. **WHERE** 절을 사용하면 반환된 인스턴스 목록을 구체화할 수 있습니다.

UPDATE 또는 **DELETE** 문을 실행한 결과는 실제로 영향을 받는 행 수입니다(업데이트 또는 삭제됨).

예제: 대량 업데이트 명령

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Company set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

예제: 대량 삭제 구문

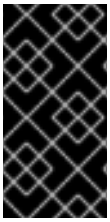
```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Company where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

Query.executeUpdate() 메서드에서 반환한 **int** 값은 작업에서 영향을 받은 데이터베이스 내의 엔터티 수를 나타냅니다.

내부적으로 데이터베이스는 여러 SQL 문을 사용하여 DML **Update** 또는 **Delete** 요청에 따라 작업을 실행할 수 있습니다. 이는 테이블과 업데이트 또는 삭제해야 하는 조인 테이블 사이에 존재하는 관계 때문일 수 있습니다.

예를 들어 위의 예와 같이 delete 문을 실행하면 **oldName** 으로 이름이 지정된 회사의회사 테이블뿐만 아니라 조인 테이블에 대해서도 삭제가 실행될 수 있습니다. 따라서 이전 예제 가 성공적으로 실행된 결과, **Employee** 테이블과의 양방향, 다대다 관계의 회사 테이블도 해당 조인 테이블인**Company_Employee** 에서 행이 손실됩니다.

deletedEntries 값에는 조인 테이블의 행을 포함하여 이 작업으로 인해 영향을 받는 모든 행의 수가 포함됩니다.



중요

활성 지속성 컨텍스트에서 데이터베이스와 엔터티 간에 불일치가 발생할 수 있으므로 대규모 업데이트 또는 삭제 작업을 실행할 때는 주의해야 합니다. 일반적으로 대규모 업데이트 및 삭제 작업은 새 지속성 컨텍스트의 트랜잭션 내에서 또는 해당 작업의 상태에 영향을 줄 수 있는 엔터티를 가져오거나 액세스하기 전에만 수행해야 합니다.

INSERT 문 정보

HQL은 **INSERT** 문을 정의할 수 있는 기능을 추가합니다. 여기에 해당하는 Jakarta Persistence 쿼리 언어는 없습니다. HQL **INSERT** 문의 Backus-Naur Form (BNF)은 다음과 같습니다.

```
insert_statement ::= insert_clause select_statement

insert_clause ::= INSERT INTO entity_name (attribute_list)

attribute_list ::= state_field[, state_field ]*
```

attribute_list 는 SQL **INSERT** 문의 열 사양과 유사합니다. 매핑된 상속과 관련된 엔터티의 경우 명명된 엔터티에 직접 정의된 속성만 **attribute_list** 에서 사용할 수 있습니다. 슈퍼 클래스 속성은 허용되지 않으며 하위 클래스 속성은 의미가 없습니다. 즉, **INSERT** 문은 본질적으로 무형식입니다.

주의

select_statement 는 유효한 HQL 선택 쿼리일 수 있으며 반환 유형이 삽입에서 예상되는 유형과 일치해야 한다는 점에 유의하십시오. 현재는 검사를 데이터베이스에 재귀하는 것이 아니라 쿼리 컴파일 중에 확인됩니다. 이로 인해 *동일한 것과* 반대되는 Hibernate 유형에 문제가 발생할 수 있습니다. 예를 들어, 데이터베이스가 구분되지 않거나 변환을 처리할 수 없는 경우에도 **org.hibernate.type.DateType** 으로 매핑된 속성과 **org.hibernate.type.TimestampType** 으로 정의된 속성 간에 일치하지 않는 문제가 발생할 수 있습니다.

id 속성의 경우 insert 문은 두 가지 옵션을 제공합니다.**attribute_list** 에서 **id** 속성을 명시적으로 지정할 수 있습니다. 이 경우 해당 값이 해당 선택 표현식에서 가져오거나 생성된 값이 사용되는 경우 **attribute_list** 에서 생략할 수 있습니다. 이 후자의 옵션은 "데이터베이스에서"를 작동하는 **id** 생성기를 사용하는 경우에만 사용할 수 있습니다. 이 옵션을 "메모리" 유형 생성기와 함께 사용하면 구문 분석 중에 예외가 발생합니다.

최적화된 잠금 속성의 경우 삽입 문은 다시 두 가지 옵션을 제공합니다. 해당 선택 표현식에서 값을 가져오는 경우 **attribute_list** 의 속성을 지정하거나 **attribute_list** 에서 생략할 수 있습니다. 이 경우 해당 **org.hibernate.type.VersionType** 에 의해 정의된 시드 값이 사용됩니다.

예제: INSERT 쿼리 설명

```
String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer
c where ...";
int createdEntities = s.createQuery(hqlInsert).executeUpdate();
```

예제: 대량 삽입문

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into Account (id, name) select c.id, c.name from Customer c where
...";
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

SELECT 문을 사용하여 **id** 속성 값을 제공하지 않으면 기본 데이터베이스에서 자동 생성된 키를 지원하는 한 식별자가 생성됩니다. 이 대규모 삽입 작업의 반환 값은 데이터베이스에 실제로 생성된 항목 수입니다.

4.3. HQL 주문 정보

쿼리 결과도 주문할 수 있습니다. **ORDER BY** 절은 결과를 주문하는 데 사용할 선택한 값을 지정하는 데 사용됩니다. order-by 절의 일부로 유효한 것으로 간주되는 식 유형은 다음과 같습니다.

- 상태 필드
- 구성 요소/임베드 가능 특성
- 산술 연산, 함수 등과 같은 스칼라 표현식.
- 이전 표현식 유형의 select 절에 선언된 ID 변수

HQL은 order-by 절에서 참조하는 모든 값을 select 절에 지정해야 하지만 Jakarta Persistence 쿼리 언어에 필요합니다. 데이터베이스 이식성을 지연하는 애플리케이션은 select 절에서 참조하지 않는 order-by 절에서 값을 참조하는 것을 일부 데이터베이스에서 지원하지는 않는다는 점을 알고 있어야 합니다.

주문에 포함된 개별 표현식을 **ASC** (가져오기) 또는 **DESC** (내림차순)와 함께 사용하여 원하는 순서 방향을 나타낼 수 있습니다.

예제: 주문 기준

```
// legal because p.name is implicitly part of p
select p
from Person p
order by p.name

select c.id, sum( o.total ) as t
from Order o
```

```
    inner join o.customer c
group by c.id
order by t
```

4.4. 컬렉션 멤버 레퍼런스 정보

컬렉션 값 연결에 대한 참조는 실제로 해당 컬렉션의 *값을* 나타냅니다.

예제: 수집 참조

```
select c
from Customer c
    join c.orders o
    join o.lineItems l
    join l.product p
where o.status = 'pending'
  and p.status = 'backorder'

// alternate syntax
select c
from Customer c,
    in(c.orders) o,
    in(o.lineItems) l
    join l.product p
where o.status = 'pending'
  and p.status = 'backorder'
```

이 예제에서 식별 변수 o 는 실제로 Customer#orders 연관 요소의 유형인 개체 모델 유형 Order를 나타냅니다.

예제에서는 IN 구문을 사용하여 컬렉션 연관 조인을 지정하는 대체 구문도 보여줍니다. 두 형식 모두 동일합니다. 애플리케이션이 사용하기 위해 선택하는 양식은 단순히 맛보기 때문일 뿐입니다.

4.5. 정규화된 경로 표현식 정보

이전에 수집 가치 있는 연관이 해당 컬렉션의 *값을* 참조한다고 설명되었습니다. 컬렉션 유형에 따라 명시적 자격 표현식 집합도 사용할 수 있습니다.

표 4.2. 유효한 경로 표현식

표현식	설명
값	collection 값을 참조합니다. 한정자를 지정하지 않는 것과 동일합니다. 의도를 명시적으로 표시하는 데 유용합니다. 모든 유형의 컬렉션 값 참조에 유효합니다.

표현식	설명
인덱스	HQL 규칙에 따라 javax.persistence.OrderColumn 주석을 지정하는 Maps 및 List 둘 다에서 유효하여 Map 키 또는 List 위치(OrderColumn 값이라고 함)를 참조합니다. 그러나 Jakarta Persistence 쿼리 언어는 목록 사례에서 사용할 수 있도록 예약하고 MAP 사례에 대해 **KEY** 를 추가합니다. 자카르타 지속성 공급자 이식성에 관심이 있는 애플리케이션은 이러한 차이점을 알고 있어야 합니다.
KEY	맵에만 유효합니다. 맵의 키를 나타냅니다. 키가 엔터티인 경우 를 추가로 탐색할 수 있습니다.
항목	맵에만 유효합니다. 맵의 논리적 java.util.Map.Entry authenticateple(키와 값의 조합)을 나타냅니다. **ENTRY** 는 터미널 경로로만 유효하며 select 절에만 유효합니다.

예제: 정규화된 컬렉션 참조

```
// Product.images is a Map<String,String> : key = a name, value = file path

// select all the image file paths (the map value) for Product#123
select i
from Product p
    join p.images i
where p.id = 123

// same as above
select value(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names (the map key) for Product#123
select key(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names and file paths (the 'Map.Entry') for Product#123
select entry(i)
from Product p
    join p.images i
where p.id = 123

// total the value of the initial line items for all orders for a customer
select sum( li.amount )
from Customer c
        join c.orders o
```

```
        join o.lineItems li
where c.id = 123
  and index(li) = 1
```

4.6. HQL 기능 정보

HQL은 사용 중인 기본 데이터베이스와 관계없이 사용할 수 있는 몇 가지 표준 기능을 정의합니다. HQL은 전화 번호 및 애플리케이션에 정의된 추가 기능도 이해할 수 있습니다.

4.6.1. HQL Standardized Functions 정보

사용 중인 기본 데이터베이스와 관계없이 HQL에서 다음 기능을 사용할 수 있습니다.

표 4.3. HQL 표준 기능

함수	설명
BIT_LENGTH	바이너리 데이터의 길이를 반환합니다.
CAST	SQL 플러시 수행. 드리프트 대상은 사용할 Hibernate 매핑 유형의 이름을 지정해야 합니다.
EXTRACT	datetime 값에 SQL 압축을 수행합니다. 추출 날짜/시간 값의 일부를 반환합니다(예: 연도). 아래에서 축약된 양식을 참조하십시오.
SECOND	두 번째 추출을 위한 축약된 추출 양식.
분	분 추출을 위한 약어 추출 양식.
HOUR	시간을 추출하기 위한 약어 추출 양식.
DAY	오늘의 추출을 위한 약어 추출 양식.
월	월 추출을 위한 약어 추출 양식.
년	연도 추출을 위한 약어 추출 양식.
STR	문자 데이터로 값을 나타내는 약어 형식입니다.

4.6.2. HQL 비표준 함수 정보

Hibernate callects는 특정 데이터베이스 제품에 사용할 수 있는 것으로 알려진 추가 기능을 등록할 수 있습니다. 데이터베이스를 사용하거나 전화하는 경우에만 사용할 수 있습니다. 데이터베이스 이식성을 목표로 하는 애플리케이션은 이 카테고리의 기능을 사용하지 않아야 합니다.

애플리케이션 개발자는 자체 기능 세트도 제공할 수 있습니다. 일반적으로 SQL 코드 조각에 대한 사용자 지정 SQL 함수 또는 별칭을 나타냅니다. 이러한 기능 선언은 org.hibernate.cfg.Configuration의 addSqlœction 방법을 사용하여 수행됩니다.

4.6.3. 연결 작업 정보

HQL은 연결(ConCAT) 기능을 지원하는 것 외에도 연결 연산자를 정의합니다. 이는 Jakarta Persistence 쿼리 언어에 의해 정의되지 않으므로 이식 가능한 애플리케이션은 사용하지 않아야 합니다. 연결 운영자는 SQL 연결 연산자(||)에서 가져옵니다.

예제: 연결 작업 예

```
select 'Mr. ' || c.name.first || ' ' || c.name.last
from Customer c
where c.gender = Gender.MALE
```

4.7. 동적 인스턴스화 정보

select 절에만 유효한 특정 표현식 유형이 있습니다. Hibernate를 이 "동적 인스턴스화"라고 합니다. Jakarta Persistence 쿼리 언어는 이 기능 중 일부를 지원하고 "건설자 표현식"이라고 합니다.

예제: 동적 인스턴스화 예 - 빌드자

```
select new Family( mother, mate, offspr )
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

따라서 여기에서 Object[]를 처리하는 대신 쿼리 결과로 반환될 타입의 안전한 Java 오브젝트의 값을 래핑합니다. 클래스 참조는 정규화되어야 하며 일치하는 생성자가 있어야 합니다.

여기에서는 클래스를 매핑할 필요가 없습니다. 엔터티를 나타내는 경우 결과 인스턴스가 NEW 상태로 반환됩니다(관리되지 않음).

이는 Jakarta Persistence 쿼리 언어도 지원합니다. HQL은 추가 "동적 인스턴스화" 기능을 지원합니다. 먼저 쿼리는 스칼라 결과에 대한 Object[] 대신 List를 반환하도록 지정할 수 있습니다.

예제: 동적 인스턴스화 예 - 목록

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

이 쿼리의 결과는 List<Object[]>와 달리 List<List>가 됩니다.

HQL은 스칼라 결과를 래핑하는 기능도 지원합니다.

예제: 동적 인스턴스화 예 - 맵

```
select new map( mother as mother, offspr as offspr, mate as mate )
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr

select new map( max(c.bodyWeight) as max, min(c.bodyWeight) as min, count(*) as n )
from Cat cxt
```

이 쿼리의 결과는 List<Object[]>와 달리 List<Map<String,Object✓이 됩니다. 맵의 키는 선택 표현식에 지정된 별칭으로 정의됩니다.

4.8. HQL 서술자 정보

서술자는 **where** 절, have 절, 검색된 대소문자 표현식의 기반을 형성합니다. NULL 값을 포함하는 부울 비교는 일반적으로 **TRUE** 또는 **FALSE** 로 해석되지만 일반적으로 TRUE 또는 FALSE로 해석되는 표현식입니다.

HQL 서술자

- null 서술자
  null 값을 확인합니다. 기본 속성 참조, 엔터티 참조 및 매개 변수에 적용할 수 있습니다. HQL을 사용하면 구성 요소/embeddable 유형에 적용할 수 있습니다.

  예제: Null 확인

  ```
  // select everyone with an associated address
  select p
  from Person p
  where p.address is not null

  // select everyone without an associated address
  select p
  from Person p
    where p.address is null
  ```

- 서술자처럼
  문자열 값에 대해 유사한 비교 수행. 구문은 다음과 같습니다.

  ```
  like_expression ::=
        string_expression
        [NOT] LIKE pattern_value
        [ESCAPE escape_character]
  ```

  의미 체계는 표현식과 같은 SQL의 내용을 따릅니다. **pattern_value** 는 **string_ expressionion**에서 일치시킬 패턴입니다. SQL과 마찬가지로 **pattern_value** 는 _ (underscore) 및 % (percent)를 와일드카드로 사용할 수 있습니다. 의미는 동일합니다. 단일 문자 와 일치 (_) % 는 임의 수의 문자와 일치합니다.

  선택 사항인 **escape_character** 는 **pattern_value** 에서 특수한 의미를 _ 및 % 로 이스케이프하는 데 사용되는 이스케이프 문자를 지정하는 데 사용됩니다. 이는 _ 또는 % 를 포함한 패턴을 검색해야 할 때 유용합니다.

  예제: LIKE 서술자

  ```
  select p
  from Person p
  where p.name like '%Schmidt'

  select p
  from Person p
  where p.name not like 'Jingleheimmer%'
  ```

```
// find any with name starting with "sp_"
select sp
from StoredProcedureMetadata sp
where sp.name like 'sp|_%' escape '|'
```

- 서술자 사이
  SQLWEEN 표현식 과 유사합니다. 값이 다른 2개 값 범위 내에 있는 평가를 수행합니다. 모든 피연산자가 비슷한 유형을 가져야 합니다.

  예제: 수요일 서술자

```
select p
from Customer c
   join c.paymentHistory p
where c.id = 123
  and index(p) between 0 and 9

select c
from Customer c
where c.president.dateOfBirth
      between {d '1945-01-01'}
          and {d '1965-01-01'}

select o
from Order o
where o.total between 500 and 5000

select p
from Person p
where p.name between 'A' and 'E'
```

- 서술자
  IN 서술자는 특정 값이 값 목록에 있는지 확인합니다. 구문은 다음과 같습니다.

```
in_expression ::= single_valued_expression
          [NOT] IN single_valued_list

single_valued_list ::= constructor_expression |
          (subquery) |
          collection_valued_input_parameter

constructor_expression ::= (expression[, expression]*)
```

  single_valued_expression 의 유형과 single_valued_list 의 개별 값이 일관되어야 합니다. Jakarta Persistence 쿼리 언어는 여기에서 유효한 유형을 문자열, 숫자, 날짜, 시간, 타임스탬프 및 열거 유형으로 제한합니다. 자카르타 지속성 쿼리 언어 에서는 **single_valued_expression** 은 다음을 참조할 수 있습니다.

  - 단순 속성의 용어인 "상태 필드". 특히, 연결 및 구성 요소/임베디드 속성이 제외됩니다.
  - 엔터티 유형 표현식.
    HQL에서 **single_valued_expression** 은 훨씬 광범위한 표현식 유형을 참조할 수 있습니다. 단일 값 연결은 허용됩니다. 이러한 기능은 기본 데이터베이스에서 "로열 값 생성자 구문"에 대한 지원 수준에 따라 달라지지만 구성 요소/임베디드 속성입니다. 또한 HQL은 어떤 방식으로도

값 유형을 제한하지 않지만, 애플리케이션 개발자는 다양한 유형의 경우 기본 데이터베이스 벤더에 따라 제한된 지원이 발생할 수 있다는 점을 알고 있어야 합니다. 이는 주로 Jakarta Persistence 쿼리 언어 제한의 이유입니다.

값 목록은 다양한 소스에서 가져올 수 있습니다. **constructor_Expression** 및 **collection_valued_input_parameter** 에서는 값 목록이 비어 있으면 안 됩니다. 값은 하나 이상 포함되어야 합니다.

예제: 서술자

```
select p
from Payment p
where type(p) in (CreditCardPayment, WireTransferPayment)

select c
from Customer c
where c.hqAddress.state in ('TX', 'OK', 'LA', 'NM')

select c
from Customer c
where c.hqAddress.state in ?

select c
from Customer c
where c.hqAddress.state in (
    select dm.state
    from DeliveryMetadata dm
    where dm.salesTax is not null
)

// Not Jakarta Persistence query language compliant!
select c
from Customer c
where c.name in (
    ('John','Doe'),
    ('Jane','Doe')
)

// Not Jakarta Persistence query language compliant!
select c
from Customer c
where c.chiefExecutive in (
    select p
    from Person p
    where ...
)
```

4.9. 관계형 비교 정보

비교에는 =, >, >=, <, KEY, <>와 같은 비교 연산자 중 하나가 포함됩니다. HQL은 또한 <>과 동의한 비교 연산자로 !=를 정의합니다. 피연산자는 동일한 유형이어야 합니다.

예제: 관계 비교 예

```
// numeric comparison
```

```
select c
from Customer c
where c.chiefExecutive.age < 30

// string comparison
select c
from Customer c
where c.name = 'Acme'

// datetime comparison
select c
from Customer c
where c.inceptionDate < {d '2000-01-01'}

// enum comparison
select c
from Customer c
where c.chiefExecutive.gender = com.acme.Gender.MALE

// boolean comparison
select c
from Customer c
where c.sendEmail = true

// entity type comparison
select p
from Payment p
where type(p) = WireTransferPayment

// entity value comparison
select c
from Customer c
where c.chiefExecutive = c.chiefTechnologist
```

비교에는 하위 쿼리 한정자 - **ALL,ANY,SOME** 도 포함될 수 있습니다.**SOME** 과 **ANY** 는 동의어입니다.

하위 쿼리 결과에 있는 모든 값에 대해 비교가 참이면 **ALL** 한정자가 true로 확인됩니다. 하위 쿼리 결과가 비어 있으면 false로 확인됩니다.

예제: 모든 하위 쿼리 비교 예

```
// select all players that scored at least 3 points
// in every game.
select p
from Player p
where 3 > all (
   select spg.points
   from StatsPerGame spg
   where spg.player = p
)
```

하위 쿼리 결과로 값 중 하나 이상에 대해 비교가 참이면 **ANY/SOME** 한정자가 true로 확인됩니다. 하위 쿼리 결과가 비어 있으면 false로 확인됩니다.

4.10. 바이트 코드 개선 사항

바이트 코드 개선은 지속성 관련 기능 추가와 같은 특정 목적을 위해 클래스의 바이트 코드(.class) 표현을 조작하는 데 사용됩니다. JBoss EAP 인스턴스의 경우 런타임 시 바이트 코드 개선 사항을 수행할 수 있습니다.

중요

빌드 시간 바이트 코드 개선은 JBoss EAP에서 지원 및 테스트되지 않습니다.

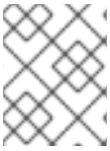
도메인 모델의 런타임 개선은 클래스 변환 수행에 JPA-defined Service Provider Interface(SPI)를 사용하므로 관리형 JPA 환경에서만 지원됩니다. 런타임 개선 사항은 기본적으로 비활성화되어 있습니다.

런타임 기능 향상을 활성화하려면 요구 사항에 따라 영구 장치에서 다음 속성 중 하나 이상의 값을 설정합니다.

- **enableLazyInitialization**: lazy 특성 로드가 필요한 경우 이 속성을 활성화합니다.
- **enableDirtyTracking**: 자체 디렉터리 추적을 위해 개선이 필요한 경우 이 속성을 활성화합니다.
- **enableAssociationManagement**: 양방향 연결 관리가 필요한 경우 이 속성을 활성화합니다.

예제: 런타임 개선 기능을 사용하도록 속성 설정

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">
   <persistence-unit name="Example">
      ...
      <properties>
         <property name="hibernate.enhancer.enableLazyInitialization" value="true" />
         <property name="hibernate.enhancer.enableDirtyTracking" value="true" />
         <property name="hibernate.enhancer.enableAssociationManagement" value="true" />
          ...
      </properties>
   </persistence-unit>
</persistence>
```



참고

런타임 기능 개선을 위해 주석이 지정된 클래스만 지원됩니다. 즉, 영구 유닛에 선언된 클래스만 향상됩니다.

4.10.1. 지연된 속성 로드

지연 속성 로드는 Hibernate에 데이터베이스에서 가져올 때 엔터티의 특정 부분만 로드해야 하고 나머지 부분도 로드해야 할 시기를 Hibernate에 알릴 수 있는 바이트 코드 개선 사항입니다. 이는 필요에 따라 엔터티 상태가 한 번에 로드되는 엔터티 중심인 지연 로드의 프록시 기반 아이디어와 다릅니다. 바이트코드 기능 향상을 통해 필요에 따라 개별 속성 또는 속성 그룹이 로드됩니다.

지연 속성은 함께 로드하도록 지정할 수 있으며 이를 *지연 그룹*이라고 합니다. 기본적으로 모든 단수 속성은 단일 그룹의 일부입니다. 하나의 지연 단일 단수 속성에 액세스하면 모든 지연 단수 속성이 로드됩니다. 지연 단일 그룹과는 달리, lazy plural 속성은 각각 개별 지연 그룹입니다. 이 동작은 **@org.hibernate.annotations.LazyGroup** 주석을 통해 명시적으로 제어할 수 있습니다.

```
@Entity
public class Customer {
```

```
    @Id
    private Integer id;

    private String name;

    @Basic( fetch = FetchType.LAZY )
    private UUID accountsPayableXrefId;

    @Lob
    @Basic( fetch = FetchType.LAZY )
    @LazyGroup( "lobs" )
    private Blob image;

    public Integer getId() {
       return id;
    }

    public void setId(Integer id) {
       this.id = id;
    }

    public String getName() {
       return name;
    }

    public void setName(String name) {
       this.name = name;
    }

    public UUID getAccountsPayableXrefId() {
       return accountsPayableXrefId;
    }

    public void setAccountsPayableXrefId(UUID accountsPayableXrefId) {
       this.accountsPayableXrefId = accountsPayableXrefId;
    }

    public Blob getImage() {
       return image;
    }

    public void setImage(Blob image) {
       this.image = image;
    }
 }
```

위의 예에는 **accountsPayableXrefId** 및 **image** 라는 두 개의 지연 속성이 있습니다. 이러한 각 특성은 다른 가져오기 그룹의 일부입니다. **accountsPayableXrefId** 속성은 기본 가져오기 그룹의 일부입니다. 즉 **accountsPayableXrefId** 에 액세스하는 경우 이미지 특성을 강제로 로드하지 않으며 그 반대의 경우도 마찬가지입니다.

4.10.2. 인라인 더티 추적

Hibernate는 지속성 컨텍스트에서 변경된 엔터티를 결정하기 위해 차이점 기반 추적 계산을 지원합니다. 즉, Hibernate는 엔티티의 마지막 알려진 상태에 대해 엔티티의 현재 상태를 확인하여 변경 사항을 추적할 수 있음을 의미합니다.

인라인 더티 추적 기능은 상태 밀도 계산을 수행하지 않고 내부 상태를 변경할 수 있는 데이터 유형을 추적하는 데 유용합니다. Hibernate는 클래스의 바이트 코드를 조작하여 더티 추적 기능을 엔터티에 직접 추가하기 때문에 엔티티가 변경된 특성을 추적할 수 있습니다.

4.10.3. BI-directal 연결 관리

Hibernate는 Java 언어의 규칙에 가까운 애플리케이션을 개발하는 데 도움이 됩니다.

다음 예제에서는 잘못된 Java 사용법을 보여줍니다.

예제: 잘못된 Java 사용

```
Order order = new Order();
LineItem lineItem = new LineItem();
order.getLineItems().add( lineItem );

// This blows up (NPE) in normal Java usage
lineItem.getOrder.getname();
```

다음 예제에서는 올바른 Java 사용법을 보여줍니다.

예제: 올바른 Java 사용

```
Order order = new Order();
LineItem lineItem = new LineItem();
order.getLineItems().add( lineItem );
lineItem.setOrder( order );

// Now this is OK...
lineItem.getOrder.getname();
```

양방향 연관 관리 기능을 사용하면 한 측이 조작될 때 양방향 연관의 다른 측면이 작동할 수 있습니다.

5장. HIBERNATE 서비스

5.1. HIBERNATE 서비스 정보

서비스는 Hibernate에 다양한 유형의 기능의 플러그형 구현을 제공하는 클래스입니다. 특히 특정 서비스 계약 인터페이스의 구현입니다. 인터페이스를 서비스 역할이라고 합니다. 구현 클래스를 서비스 구현이라고 합니다. 일반적으로 사용자는 모든 표준 서비스 역할(덮어쓰기)의 대체 구현을 연결할 수 있습니다. 또한 기본 서비스 역할 집합(확장)을 넘어 추가 서비스를 정의할 수도 있습니다.

5.2. 서비스 계약 정보

서비스의 기본 요구 사항은 마커 인터페이스 org.hibernate.service.Service를 구현하는 것입니다. Hibernate는 일부 기본 유형 안전에 이를 내부적으로 사용합니다.

선택적으로 서비스는 org.hibernate.service.spi.Startable 및 org.hibernate.service.spi.Stoppable 인터페이스를 구현하여 시작 및 중지됨 알림을 받을 수 있습니다. 또 다른 옵션 서비스 계약은 org.hibernate.service.spi.Manageable으로, Jakarta Management 통합이 활성화되면 Jakarta Management에서 서비스를 관리 가능으로 표시합니다.

5.3. 서비스 종속성 유형

서비스는 다음 접근 방법 중 하나를 사용하여 다른 서비스에 대한 종속성을 선언할 수 있습니다.

@org.hibernate.service.spi.InjectService

단일 매개 변수를 수락하고 **@InjectService** 주석을 달 수 있는 서비스 구현 클래스의 모든 메서드는 다른 서비스의 주입을 요청하는 것으로 간주됩니다.
기본적으로 메서드 매개 변수의 유형은 삽입할 서비스 역할이어야 합니다. 매개 변수 유형이 서비스 역할과 다른 경우 **InjectService** 의 **serviceRole** 특성을 사용하여 역할의 이름을 명시적으로 지정해야 합니다.

기본적으로 삽입된 서비스는 필요한 것으로 간주되며, 이는 명명된 종속 서비스가 누락된 경우 시작에 실패하는 것입니다. 주입할 서비스가 선택 사항인 경우 **InjectService** 의 필수 특성을**false** 로 선언해야 합니다. 기본값은 **true**입니다.

org.hibernate.service.spi.ServiceRegistryAwareService

두 번째 방법은 서비스에서 단일 **injectServices** 메서드를 선언하는 선택적 서비스 인터페이스 **org.hibernate.service.spi.ServiceRegistryAwareService** 를 구현하는 가져오기 접근법입니다.
시작하는 동안 Hibernate는 이 인터페이스를 구현하는 서비스에 **org.hibernate.service.ServiceRegistry** 자체를 주입합니다. 그런 다음 서비스는**ServiceRegistry** 참조를 사용하여 필요한 추가 서비스를 찾을 수 있습니다.

5.3.1. 서비스 레지스트리

5.3.1.1. ServiceRegistry 정보

서비스 자체 외에도 중앙 서비스 API는 org.hibernate.service.ServiceRegistry 인터페이스입니다. 서비스 레지스트리의 주요 목적은 서비스에 대한 액세스를 보유, 관리 및 제공하는 것입니다.

서비스 레지스트리는 계층 구조입니다. 한 레지스트리의 서비스는 해당 레지스트리와 상위 레지스트리의 서비스를 활용하고 활용할 수 있습니다.

org.hibernate.service.ServiceRegistryBuilder를 사용하여 org.hibernate.service.ServiceRegistry 인스턴스를 빌드합니다.

ServiceRegistryBuilder를 사용하여 ServiceRegistry 생성 예

```
ServiceRegistryBuilder registryBuilder =
    new ServiceRegistryBuilder( bootstrapServiceRegistry );
    ServiceRegistry serviceRegistry = registryBuilder.buildServiceRegistry();
```

5.3.2. 사용자 정의 서비스

5.3.2.1. 사용자 지정 서비스 정보

org.hibernate.service.ServiceRegistry 가 구축되면 변경 불가능한 것으로 간주됩니다. 서비스 자체에서 재구성을 수락할 수 있지만, 여기에서 불변성은 서비스 추가 또는 교체를 의미합니다. 따라서 **org.hibernate.service.ServiceRegistryBuilder** 에서 제공하는 또 다른 역할은 생성된 **org.hibernate.service.ServiceRegistry** 에 포함될 서비스를 수정할 수 있습니다.

org.hibernate.service.ServiceRegistryBuilder 에 사용자 지정 서비스에 대해 알리는 두 가지 방법이 있습니다.

- **org.hibernate.service.spi.BasicServiceInitiator** 클래스를 구현하여 서비스 클래스의 온디맨드 구성을 제어하고 **addInitiator** 메서드를 사용하여 **org.hibernate.service.ServiceRegistryBuilder** 에 추가합니다.
- 서비스 클래스를 인스턴스화하고 addService 메서드를 사용하여 **org.hibernate.service.ServiceRegistryBuilder** 에 추가합니다.

두 방법 중 하나는 새 서비스 역할 추가 등의 레지스트리 확장 및 서비스 구현 대체와 같은 서비스 재정의에 유효합니다.

예제: ServiceRegistryBuilder를 사용하여 기존 서비스를 사용자 지정 서비스로 교체

```
ServiceRegistryBuilder registryBuilder =
    new ServiceRegistryBuilder(bootstrapServiceRegistry);
registryBuilder.addService(JdbcServices.class, new MyCustomJdbcService());
ServiceRegistry serviceRegistry = registryBuilder.buildServiceRegistry();

public class MyCustomJdbcService implements JdbcServices{

  @Override
  public ConnectionProvider getConnectionProvider() {
    return null;
  }

  @Override
  public Dialect getDialect() {
    return null;
  }

  @Override
  public SqlStatementLogger getSqlStatementLogger() {
    return null;
  }
```

```
    @Override
    public SqlExceptionHelper getSqlExceptionHelper() {
        return null;
    }

    @Override
    public ExtractedDatabaseMetaData getExtractedMetaDataSupport() {
        return null;
    }

    @Override
    public LobCreator getLobCreator(LobCreationContext lobCreationContext) {
        return null;
    }

    @Override
    public ResultSetWrapper getResultSetWrapper() {
        return null;
    }
}
```

5.3.3. Boot-Strap 레지스트리

5.3.3.1. 부트스트랩 레지스트리 정보

boot-strap 레지스트리에는 대부분의 작업이 작동하려면 반드시 사용할 수 있는 서비스가 있습니다. 여기에 있는 주요 서비스는 완벽한 예인 **ClassLoaderService** 입니다. 구성 파일을 해결하더라도 리소스 조회 등 클래스 로드 서비스에 액세스할 수 있어야 합니다. 이는 일반적으로 상위 레지스트리가 아닌 루트 레지스트리입니다.

boot-strap 레지스트리 인스턴스는 **org.hibernate.service.BootstrapServiceRegistryBuilder** 클래스를 사용하여 빌드됩니다.

Using BootstrapServiceRegistryBuilder

예제: Using BootstrapServiceRegistryBuilder

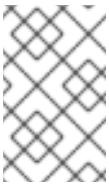
```
BootstrapServiceRegistry bootstrapServiceRegistry =
    new BootstrapServiceRegistryBuilder()
    // pass in org.hibernate.integrator.spi.Integrator instances which are not
    // auto-discovered (for whatever reason) but which should be included
    .with(anExplicitIntegrator)
    // pass in a class loader that Hibernate should use to load application classes
    .with(anExplicitClassLoaderForApplicationClasses)
    // pass in a class loader that Hibernate should use to load resources
    .with(anExplicitClassLoaderForResources)
    // see BootstrapServiceRegistryBuilder for rest of available methods
    ...
    // finally, build the bootstrap registry with all the above options
    .build();
```

5.3.3.2. BootstrapRegistry Services

org.hibernate.service.classloading.spi.ClassLoaderService

Hibernate는 클래스 로더와 상호 작용해야 합니다. 그러나 Hibernate 또는 모든 라이브러리에서 클래스 로더와 상호 작용하는 방식은 애플리케이션을 호스팅하는 런타임 환경에 따라 달라집니다. 애플리케이션 서버, OSGi 컨테이너 및 기타 모듈식 클래스 로드 시스템은 매우 구체적인 클래스 로드 요구 사항을 적용합니다. 이 서비스는 Hibernate에 환경 복잡성을 추상화하는 기능을 제공합니다. 그리고 중요한 사실은 하나의 구성 요소로 이루어집니다.
클래스 로더와 상호 작용하는 경우 Hibernate에는 다음과 같은 기능이 필요합니다.

- 애플리케이션 클래스를 찾을 수 있는 기능
- 통합 클래스를 찾을 수 있는 기능
- 속성 파일 및 XML 파일과 같은 리소스를 찾는 기능
- **java.util.ServiceLoader**로드 기능



참고

현재 애플리케이션 클래스를 로드하고 통합 클래스를 로드하는 기능이 서비스의 단일 부하 클래스 기능으로 결합되어 있습니다. 이후 릴리스에서 변경될 수 있습니다.

org.hibernate.integrator.spi.IntegratorService

애플리케이션, 애드온 및 기타 모듈은 Hibernate와 통합되어야 합니다. 이전 접근 방식에는 각 개별 모듈의 등록을 조정하기 위해 구성 요소(일반적으로 애플리케이션이 필요했습니다. 이 등록은 각 모듈의 통합자를 대신하여 수행되었습니다.
이 서비스는 검색 측면에 중점을 둡니다. **org.hibernate.service.classloading.spi.ClassLoaderService**에서 제공하는 표준 **java.util.ServiceLoader** 기능을 활용하여 **org.hibernate.integrator.spi.Integrator** 계약의 구현을 검색합니다.

통합자는 **/META-INF/services/org.hibernate.integrator.spi.Integrator** 라는 파일을 정의하고 클래스 경로에서 사용할 수 있도록 합니다.

이 파일은 **java.util.ServiceLoader** 메커니즘에서 사용합니다. 행당 하나씩 **org.hibernate.integrator.spi.Integrator** 인터페이스를 구현하는 정규화된 클래스를 나열합니다.

5.3.4. 세션팩트 레지스트리

모든 레지스트리 유형의 인스턴스를 지정된 **org.hibernate.SessionFactory** 를 대상으로 처리하는 것이 좋지만 이 그룹의 서비스 인스턴스는 단일 **org.hibernate.SessionFactory** 에 명시적으로 속합니다.

차이점은 시작되어야 하는 시기의 문제에 해당합니다. 일반적으로 시작하려면 **org.hibernate.SessionFactory** 에 액세스할 수 있어야 합니다. 이 특수 레지스트리는 **org.hibernate.service.spi.SessionFactoryServiceRegistry** 입니다.

5.3.4.1. SessionFactory 서비스

org.hibernate.event.service.spi.EventListenerRegistry

설명

이벤트 리스너를 관리하는 서비스입니다.

개시자

org.hibernate.event.service.internal.EventListenerServiceInitiator

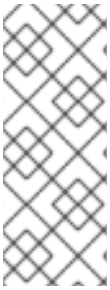
구현

org.hibernate.event.service.internal.EventListenerRegistryImpl

5.3.5. 통합업체

org.hibernate.integrator.spi.Integrator 는 개발자가 작동하는**SessionFactory** 를 구축하는 프로세스에 후크할 수 있는 간단한 방법을 제공하기 위한 것입니다. **org.hibernate.integrator.spi.Integrator** 인터페이스는 두 가지 관심 방법을 정의합니다.

- 통합을 통해 구축 프로세스로 전환할 수 있습니다.
- 해체 하면 **SessionFactory** 종료를 시작할 수 있습니다.



참고

과부하 형태인 **org.hibernate.integrator.spi.Integrator** 에 정의된 세 번째 방법은 **org.hibernate.metamodel.source.MetadataImplementor** 대신 **org.hibernate.cfg.Configuration**을 수락합니다.

IntegratorService에서 제공하는 검색 접근 방식 외에도 애플리케이션은 **BootstrapService Registry** 를 빌드할 때 Integrator 구현을 수동으로 등록할 수 있습니다.

5.3.5.1. 통합업체 사용 사례

org.hibernate.integrator.spi.Integrator 의 주요 사용 사례는 이벤트 리스너를 등록하고 서비스를 제공하고 **org.hibernate.integrator.spi.ServiceContributingIntegrator** 를 참조하십시오.

예제: 이벤트 리스너 등록

```
public class MyIntegrator implements org.hibernate.integrator.spi.Integrator {

    public void integrate(
            Configuration configuration,
            SessionFactoryImplementor sessionFactory,
            SessionFactoryServiceRegistry serviceRegistry) {
        // As you might expect, an EventListenerRegistry is the thing with which event listeners
are registered  It is a
        // service so we look it up using the service registry
        final EventListenerRegistry eventListenerRegistry =
serviceRegistry.getService(EventListenerRegistry.class);

        // If you wish to have custom determination and handling of "duplicate" listeners, you
would have to add an
        // implementation of the org.hibernate.event.service.spi.DuplicationStrategy contract like
this
        eventListenerRegistry.addDuplicationStrategy(myDuplicationStrategy);

        // EventListenerRegistry defines 3 ways to register listeners:
        //     1) This form overrides any existing registrations with
        eventListenerRegistry.setListeners(EventType.AUTO_FLUSH,
myCompleteSetOfListeners);
        //     2) This form adds the specified listener(s) to the beginning of the listener chain
        eventListenerRegistry.prependListeners(EventType.AUTO_FLUSH,
myListenersToBeCalledFirst);
        //     3) This form adds the specified listener(s) to the end of the listener chain
```

```
        eventListenerRegistry.appendListeners(EventType.AUTO_FLUSH,
myListenersToBeCalledLast);
    }
}
```

6장. HIBERNATE ENVERS

6.1. HIBERNATE ENVERS 정보

Hibernate Envers는 감사 및 버전 관리 시스템으로, JBoss EAP에 지속적 클래스에 대한 이전 변경 사항을 추적할 수 있습니다. 감사 테이블은 엔터티에 대한 변경 기록을 저장하는 **@Audited** 로 주석이 지정된 엔터티에 대해 생성됩니다. 그런 다음 데이터를 검색하고 쿼리할 수 있습니다.

개발자는 Envers를 통해 다음을 수행할 수 있습니다.

- Jakarta Persistence 사양에 정의된 모든 매핑 감사
- Jakarta Persistence 사양을 확장하는 모든 hibernate 매핑 감사
- 기본 Hibernate API를 사용하거나 사용하여 매핑된 감사 엔터티
- 버전 엔터티를 사용하여 각 버전에 대한 로그 데이터
- 기록 데이터 쿼리

6.2. 영구 클래스 감사 정보

지속 클래스 감사는 Hibernate Envers 및 **@Audited** 주석을 통해 JBoss EAP에서 수행됩니다. 주석이 클래스에 적용되면 엔터티의 버전 기록을 저장하는 테이블이 생성됩니다.

클래스를 변경할 때마다 audit 테이블에 항목이 추가됩니다. 항목에는 클래스의 변경 사항이 포함되어 있으며 버전 번호가 제공됩니다. 즉, 변경 사항을 롤백하거나 이전 버전을 볼 수 있습니다.

6.3. 감사 전략

6.3.1. 전략 감사 정보

감사 전략은 감사 정보를 유지, 쿼리 및 저장하는 방법을 정의합니다. 현재 Hibernate Envers에서는 다음 두 가지 감사 전략을 사용할 수 있습니다.

기본 감사 전략

- 이 전략은 시작 버전과 함께 감사 데이터를 유지합니다. 감사된 테이블에서 삽입, 업데이트 또는 삭제되는 각 행의 유효성을 시작 리버전과 함께 감사 테이블에 하나 이상의 행이 삽입됩니다.
- 감사 테이블의 행은 삽입 후 업데이트되지 않습니다. 감사 정보 쿼리는 하위 쿼리를 사용하여 느린 인덱스가 어렵고 감사 테이블에서 해당 행을 선택합니다.

유효 감사 전략

- 이 전략에서는 시작 버전과 감사 정보의 최종 버전을 저장합니다. 감사된 테이블에서 삽입, 업데이트 또는 삭제되는 각 행의 유효성을 시작 리버전과 함께 감사 테이블에 하나 이상의 행이 삽입됩니다.
- 동시에 이전 감사 행의 최종 버전 필드(사용 가능한 경우)가 이 버전으로 설정됩니다. 그러면 감사 정보에 대한 쿼리가 하위 쿼리 대신 *start 및 end revision 간에*사용할 수 있습니다. 즉, 추가 업데이트로 인해 감사 정보를 유지하는 속도가 조금 더 느리지만 감사 정보를 검색하는 속도가 훨씬 빠릅니다.

- 또한 추가 인덱스를 추가하여 개선할 수 있습니다.

감사에 대한 자세한 내용은 영구 클래스 감사 정보를 참조하십시오. 애플리케이션에 대한 감사 전략을 설정하려면 감사 전략 설정을 참조하십시오.

6.3.2. 감사 전략 설정

JBoss EAP에서는 다음 두 가지 감사 전략을 지원합니다.

- 기본 감사 전략
- 유효 감사 전략

감사 전략 정의
애플리케이션의 **persistence .xml** 파일에 **org.hibernate.envers. audit_strategy** 속성을 구성합니다. **persistence.xml** 파일에 속성이 설정되지 않은 경우 기본 감사 전략이 사용됩니다.

기본 감사 전략 설정

```
<property name="org.hibernate.envers.audit_strategy"
value="org.hibernate.envers.strategy.DefaultAuditStrategy"/>
```

Validity 감사 전략 설정

```
<property name="org.hibernate.envers.audit_strategy"
value="org.hibernate.envers.strategy.ValidityAuditStrategy"/>
```

6.3.3. Jakarta Persistence 엔터티에 감사 지원 추가

절차

JBoss EAP는 Hibernate Envers 를 통한 엔터티 감사를 사용하여 지속 클래스의 이전 변경 사항을 추적합니다. 이 섹션에서는 자카르타 지속성 엔터티에 대한 감사 지원을 추가하는 방법에 대해 설명합니다.

Jakarta Persistence 엔터티에 감사 지원 추가

1. 배포에 맞게 사용 가능한 감사 매개변수를 구성합니다. 자세한 내용은 Configure Envers Parameters 를 참조하십시오.
2. 감사할 Jakarta Persistence 엔터티를 엽니다.
3. **org.hibernate.envers.Audited** 인터페이스를 가져옵니다.
4. 감사할 각 필드 또는 속성에 **@Audited** 주석을 적용하거나 전체 클래스에 한 번 적용합니다.

   예제: 두 필드 감사

```
import org.hibernate.envers.Audited;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
```

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private int id;

    @Audited
    private String name;

    private String surname;

    @ManyToOne
    @Audited
    private Address address;

    // add getters, setters, constructors, equals and hashCode here
}
```

예제: 전체 클래스 감사

```
import org.hibernate.envers.Audited;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;

@Entity
@Audited
public class Person {
    @Id
    @GeneratedValue
    private int id;

    private String name;

    private String surname;

    @ManyToOne
    private Address address;

    // add getters, setters, constructors, equals and hashCode here
}
```

감사를 위해 Jakarta Persistence 엔터티가 구성되면 기록 변경 사항을 저장하기 위해 _AUD 라는 테이블이 생성됩니다.

6.4. 설정

6.4.1. 인버스 매개 변수 구성

JBoss EAP는 Hibernate Envers를 통해 엔터티 감사를 사용하여 영구 클래스의 이전 변경 사항을 추적합니다.

사용 가능한 Envers 매개변수 구성

1. 애플리케이션에 대한 **persistence.xml** 파일을 엽니다.
2. 필요에 따라 Envers 속성을 추가, 제거 또는 구성합니다. 사용 가능한 속성 목록은 구성 속성 삽입을 참조하십시오.

   예제: 인버스 매개 변수

```
<persistence-unit name="mypc">
  <description>Persistence Unit.</description>
  <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="create-drop" />
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.cache.use_second_level_cache" value="true" />
    <property name="hibernate.cache.use_query_cache" value="true" />
    <property name="hibernate.generate_statistics" value="true" />
    <property name="org.hibernate.envers.versionsTableSuffix" value="_V" />
    <property name="org.hibernate.envers.revisionFieldName" value="ver_rev" />
  </properties>
</persistence-unit>
```

6.4.2. 런타임 시 감사 활성화 또는 비활성화

런타임 시 엔터티 버전 감사 활성화 또는 비활성화

1. **AuditEventListener** 클래스의 하위 클래스입니다.
2. Hibernate 이벤트에서 호출되는 다음 메서드를 재정의합니다.
   - **onPostInsert**
   - **onPostUpdate**
   - **onPostDelete**
   - **onPreUpdateCollection**
   - **onPreRemoveCollection**
   - **onPostRecreateCollection**
3. 하위 클래스를 이벤트 리스너로 지정합니다.
4. 변경 사항을 감사해야 하는지 확인합니다.
5. 변경 감사를 수행해야 하는 경우 호출을 슈퍼 클래스로 전달합니다.

6.4.3. 조건부 감사 구성

Hibernate Envers는 일련의 이벤트 리스너를 사용하여 다양한 Hibernate 이벤트에 대한 감사 데이터를 유지합니다. Envers JAR이 클래스 경로에 있는 경우 이러한 리스너는 자동으로 등록됩니다.

조건부 감사 구현

1. **persistence .xml** 파일에서 **hibernate.listeners.envers.autoRegister** Hibernate 속성을 false로 설정합니다.

2. 하위 클래스 각 이벤트 리스너를 재정의합니다. 조건부 감사 논리를 하위 클래스에 배치하고 감사를 수행해야 하는 경우 수퍼 메서드를 호출합니다.

3. **org.hibernate. envers.event.EnversIntegrator**와 유사하게 **org.hibernate.integrator.spi.Integrator** 의 사용자 정의 구현을 만듭니다. 기본 클래스가 아닌 2단계로 생성된 이벤트 리스너 하위 클래스를 사용합니다.

4. **META-INF/services/org.hibernate.integrator.spi.Integrator** 파일을 JAR에 추가합니다. 이 파일에는 인터페이스를 구현하는 클래스의 정규화된 이름이 포함되어야 합니다.

6.4.4. 구성 속성 연결

표 6.1. 엔터티 데이터 버전 설정 매개변수

속성 이름	기본값	설명
org.hibernate.envers.audit_table_prefix		감사 정보를 보유할 엔터티의 이름을 생성하기 위한 감사 엔터티 이름에 앞에 오는 문자열입니다.
org.hibernate.envers.audit_table_suffix	_AUD	감사 정보를 보유할 엔터티의 이름을 생성하기 위해 감사된 엔터티 이름에 추가되는 문자열입니다. 예를 들어 테이블 이름이 **Person** 인 엔터티를 감사하면 Envers는 **Person_AUD** 라는 테이블을 생성하여 기록 데이터를 저장합니다.
org.hibernate.envers.revision_field_name	REV	버전 번호를 보유한 감사 엔터티의 필드 이름입니다.
org.hibernate.envers.revision_type_field_name	구조 유형	버전 유형을 보유한 감사 엔터티의 필드 이름입니다. 현재 가능한 리버전 유형은 각각 삽입, 수정 또는 삭제를 위한 **add**,**mod** 및 **del** 입니다.
org.hibernate.envers.revision_on_collection_change	true	이 속성은 소유하지 않은 관계 필드가 변경되지 않은 경우 리버전을 생성해야 하는지 여부를 결정합니다. 이는 일대다 관계의 컬렉션이거나 일대일 관계에서 **mappedBy** 속성을 사용하는 필드일 수 있습니다.
org.hibernate.envers.do_not_audit_optimistic_locking_field	true	true인 경우 최적화된 잠금에 사용되는 속성 **(@Version으로 주석이 추가됨**)은 감사에서 자동으로 제외됩니다.

속성 이름	기본값	설명
org.hibernate.envers.store_data_at_delete	false	이 속성은 ID 대신 엔터티 데이터를 삭제할 때 null로 표시된 다른 모든 속성과 함께 엔터티 데이터를 버전에 저장할지 여부를 정의합니다. 데이터가 마지막 버전에 있으므로 일반적으로 필요하지는 않습니다. 그러나 마지막 버전에서 액세스하는 것이 더 쉽고 효율적일 수 있습니다. 그러나 이는 삭제 전에 포함된 엔터티가 두 번 저장된다는 것을 의미합니다.
org.hibernate.envers.default_schema	null(일반 테이블과 동일)	감사 테이블에 사용되는 기본 스키마 이름입니다. **@AuditTable(schema="...")** 주석. 없는 경우 스키마는 일반 테이블의 스키마와 동일합니다.
org.hibernate.envers.default_catalog	null(일반 테이블과 동일)	감사 테이블에 사용해야 하는 기본 카탈로그 이름입니다. **@AuditTable(catalog="...")** 주석. 없는 경우 카탈로그는 일반 테이블의 카탈로그와 동일합니다.
org.hibernate.envers.audit_strategy	**org.hibernate.envers.strategy.DefaultAuditStrategy**	이 속성은 감사 데이터를 유지할 때 사용해야 하는 감사 전략을 정의합니다. 기본적으로 엔터티가 수정된 버전만 저장됩니다. 또는 **org.hibernate.envers.strategy.ValidityAuditStrategy** 는 시작 버전과 최종 버전을 모두 저장합니다. 이러한 값은 감사 행이 유효한 시기를 정의합니다.
org.hibernate.envers.audit_strategy_validity_end_rev_field_name	취소	감사 엔터티의 최종 버전 번호를 보유할 열 이름입니다. 이 속성은 유효 감사 전략을 사용하는 경우에만 유효합니다.
org.hibernate.envers.audit_strategy_validity_store_revend_timestamp	false	이 속성은 데이터가 마지막으로 유효한 버전인 최종 버전의 타임스탬프를 최종 버전 자체 외에 저장해야 하는지 여부를 정의합니다. 이 기능은 테이블 파티션을 사용하여 관계형 데이터베이스에서 이전 감사 레코드를 제거할 수 있는 데 유용합니다. 파티셔닝에는 테이블 내에 있는 열이 필요합니다. 이 속성은 **ValidityAuditStrategy** 가 사용되는 경우에만 평가됩니다.
org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name	REVEND_TSTMP	데이터가 계속 유효한 시점의 최종 버전 타임스탬프의 열 이름입니다. **ValidityAuditStrategy** 를 사용하고 **org.hibernate.envers.audit_strategy_validity_store_revend_timestamp** 가 true로 평가되는 경우에만 사용됩니다.

6.5. 감사 정보 쿼리

6.5.1. 쿼리를 통해 감사 정보 검색

Hibernate Envers는 쿼리를 통해 감사 정보를 검색하는 기능을 제공합니다.

참고

감사된 데이터에 대한 쿼리는 상관 관계가 있는 하위 선택 항목이 포함되어 있으므로 실시간 데이터에 대한 해당 쿼리보다 훨씬 느립니다.

지정된 버전에서 클래스 엔티티 쿼리

이 유형의 쿼리의 진입점은 다음과 같습니다.

```
AuditQuery query = getAuditReader()
   .createQuery()
   .forEntitiesAtRevision(MyEntity.class, revisionNumber);
```

그런 다음 **AuditEntity** 팩토리 클래스를 사용하여 제약 조건을 지정할 수 있습니다. 아래 쿼리는**name** 속성이 **John** 과 같은 엔티티만 선택합니다.

```
query.add(AuditEntity.property("name").eq("John"));
```

아래 쿼리는 지정된 엔터티와 관련된 엔터티만 선택합니다.

```
query.add(AuditEntity.property("address").eq(relatedEntityInstance));
// or
query.add(AuditEntity.relatedId("address").eq(relatedEntityId));
```

그런 다음 결과를 주문하고 제한할 수 있으며 집계 및 예측(그룹 지정 제외)을 설정할 수 있습니다. 아래 예는 전체 쿼리입니다.

```
List personsAtAddress = getAuditReader().createQuery()
   .forEntitiesAtRevision(Person.class, 12)
   .addOrder(AuditEntity.property("surname").desc())
   .add(AuditEntity.relatedId("address").eq(addressId))
   .setFirstResult(4)
   .setMaxResults(2)
   .getResultList();
```

지정된 클래스의 엔터티가 변경된 버전 쿼리

이 유형의 쿼리의 진입점은 다음과 같습니다.

```
AuditQuery query = getAuditReader().createQuery()
   .forRevisionsOfEntity(MyEntity.class, false, true);
```

이전 예제와 동일한 방식으로 이 쿼리에 제약 조건을 추가할 수 있습니다. 이 쿼리에는 다음과 같은 추가 가능성이 있습니다.

AuditEntity.revisionNumber()

audited 엔터티가 수정된 버전 번호에 제약 조건, 예측 및 순서를 지정합니다.

AuditEntity.revisionProperty(propertyName)

audited 엔터티가 수정된 버전에 해당하는 revision 엔터티의 속성에 제약 조건, 예측 및 순서를 지정합니다.

AuditEntity.revisionType()

버전 유형에 대한 액세스를 제공합니다(ADD, MOD, DEL).

그러면 필요에 따라 쿼리 결과를 조정할 수 있습니다. 아래 쿼리는 버전 번호 42 후 entity Id ID가 변경된 MyEntity 클래스의 엔터티 가 변경된 최소 버전 번호를 선택합니다.

```
Number revision = (Number) getAuditReader().createQuery()
   .forRevisionsOfEntity(MyEntity.class, false, true)
   .setProjection(AuditEntity.revisionNumber().min())
   .add(AuditEntity.id().eq(entityId))
   .add(AuditEntity.revisionNumber().gt(42))
   .getSingleResult();
```

버전에 대한 쿼리는 특성을 최소화/최대화할 수도 있습니다. 아래 쿼리는 지정된 엔터티에 대한 realDate 값이 지정된 값보다 크지만 가능한 작은 버전을 선택합니다.

```
Number revision = (Number) getAuditReader().createQuery()
   .forRevisionsOfEntity(MyEntity.class, false, true)
   // We are only interested in the first revision
   .setProjection(AuditEntity.revisionNumber().min())
   .add(AuditEntity.property("actualDate").minimize()
      .add(AuditEntity.property("actualDate").ge(givenDate))
      .add(AuditEntity.id().eq(givenEntityId)))
   .getSingleResult();
```

minimize() 및 maximize() 메서드는 제한 조건을 추가할 수 있는 기준을 반환하며, 이러한 제약 조건을 최대화/최소화된 속성을 가진 엔터티에서 충족해야 합니다.

쿼리 생성 시 전달되는 두 개의 부울 매개 변수가 있습니다.

selectEntitiesOnly

이 매개변수는 명시적 예측이 설정되지 않은 경우에만 유효합니다.
true인 경우 쿼리 결과는 지정된 제약 조건을 충족하는 수정 시 변경된 엔터티 목록이 됩니다.
false인 경우 결과는 3개의 요소 배열 목록입니다. 첫 번째 요소는 변경된 엔터티 인스턴스가 됩니다. 두 번째는 버전 데이터를 포함하는 엔터티입니다. 사용자 지정 엔터티를 사용하지 않는 경우 이 엔터티는 DefaultRevisionEntity 의 인스턴스입니다. 세 번째 요소 배열은 버전 유형(ADD, MOD, DEL)입니다.

selectDeletedEntities

이 매개변수는 엔터티를 삭제한 리버전을 결과에 포함시켜야 하는지를 지정합니다. true인 경우 엔터티에 버전 유형 DEL 및 id를 제외한 모든 필드의 값이null 이 됩니다.

지정된 속성을 수정한 엔터티 버전 쿼리

아래 쿼리는 지정된 ID가 있는 MyEntity 의 모든 개정 사항을 반환하며, 여기서 realDate 속성이 변경되었습니다.

```
AuditQuery query = getAuditReader().createQuery()
  .forRevisionsOfEntity(MyEntity.class, false, true)
  .add(AuditEntity.id().eq(id));
  .add(AuditEntity.property("actualDate").hasChanged())
```

hasChanged 조건은 추가 기준과 결합할 수 있습니다. 아래 쿼리는 revisionNumber가 생성된 시점에 **MyEntity** 의 수평 슬라이스를 반환합니다. 이는**prop1**을 수정했지만 **prop 2** 가 아닌 버전으로 제한됩니다.

```
AuditQuery query = getAuditReader().createQuery()
  .forEntitiesAtRevision(MyEntity.class, revisionNumber)
  .add(AuditEntity.property("prop1").hasChanged())
  .add(AuditEntity.property("prop2").hasNotChanged());
```

결과 세트에는 revisionNumber보다 숫자가 낮은 버전도 포함됩니다. 즉, 이 쿼리를 "prov 1이 수정되고 **prop 2** 를 그대로 사용하여 revisionNumber에서 모든**MyEntities** 변경"으로 읽을 수 없습니다.

아래 쿼리는 **forEntitiesModifiedAtRevision** 쿼리를 사용하여 이 결과를 반환하는 방법을 보여줍니다.

```
AuditQuery query = getAuditReader().createQuery()
  .forEntitiesModifiedAtRevision(MyEntity.class, revisionNumber)
  .add(AuditEntity.property("prop1").hasChanged())
  .add(AuditEntity.property("prop2").hasNotChanged());
```

지정된 버전에서 수정된 쿼리 항목

아래 예제에서는 지정된 버전에서 수정된 엔터티에 대한 기본 쿼리를 보여줍니다. 이를 통해 지정된 버전에서 엔터티 이름 및 해당 클래스를 변경할 수 있습니다.

```
Set<Pair<String, Class>> modifiedEntityTypes = getAuditReader()
   .getCrossTypeRevisionChangesReader().findEntityTypes(revisionNumber);
```

org.hibernate.envers.CrossTypeRevisionChangesReader에서도 액세스할 수 있는 다른 많은 쿼리가 있습니다.

List<Object> findEntities(Number)

지정된 버전에서 변경된 모든 감사 엔터티의 스냅샷을(추가, 업데이트 및 제거) 반환합니다. **n+1** SQL 쿼리를 실행합니다. 여기서 **n** 은 지정된 버전 내에서 수정된 여러 다른 엔터티 클래스입니다.

List<Object> findEntities(Number, RevisionType)

수정 유형별로 필터링된 지정된 버전에서 변경된(추가, 업데이트 또는 제거) 감사된 모든 엔터티의 스냅샷을 반환합니다. **n+1** SQL 쿼리를 실행합니다. 여기서**n** 은 지정된 버전 내에서 수정된 여러 다른 엔터티 클래스입니다. Map<RevisionType, List<Object>>

findEntitiesGroupByRevisionType(Number)

수정 작업에서 그룹화된 엔터티 스냅샷 목록(예: 추가, 업데이트 또는 제거)이 포함된 맵을 반환합니다. **3n+1** SQL 쿼리를 실행합니다. 여기서**n** 은 지정된 버전 내에서 수정된 여러 다른 엔터티 클래스입니다.

6.5.2. 참조된 엔티티 속성을 사용하여 엔터티 연결 전달

참조된 엔터티의 속성을 사용하여 쿼리에서 엔터티를 이동할 수 있습니다. 이를 통해 일대일 및 다대일 연결을 쿼리할 수 있습니다.

아래 예제에서는 쿼리에서 엔터티를 탐색할 수 있는 몇 가지 방법을 보여줍니다.

- 개정 번호 1에서는 소유자가 20세 또는 주소 번호 30에 있는 자동차를 찾은 다음, 자동차에서 설정한 결과를 주문합니다.

```
List<Car> resultList = auditReader.createQuery()
        .forEntitiesAtRevision( Car.class, 1 )
        .traverseRelation( "owner", JoinType.INNER, "p" )
```

```
                .traverseRelation( "address", JoinType.INNER, "a" )
                .up().up().add( AuditEntity.disjunction().add(AuditEntity.property( "p", "age" )
                    .eq( 20 ) ).add( AuditEntity.property( "a", "number" ).eq( 30 ) ) )
                .addOrder( AuditEntity.property( "make" ).asc() ).getResultList();
```

- 버전 번호 1에서 소유자 유효 기간이 소유자 주소 번호와 같은 드라이버를 찾습니다.

```
Car result = (Car) auditReader.createQuery()
                .forEntitiesAtRevision( Car.class, 1 )
                .traverseRelation( "owner", JoinType.INNER, "p" )
                .traverseRelation( "address", JoinType.INNER, "a" )
                .up().up().add(AuditEntity.property( "p", "age" )
                    .eqProperty( "a", "number" ) ).getSingleResult();
```

- 개정 번호 1에서는 소유자가 20세이거나 소유자가 없는 모든 자동차를 찾습니다.

```
List<Car> resultList = auditReader.createQuery()
                .forEntitiesAtRevision( Car.class, 1 )
                .traverseRelation( "owner", JoinType.LEFT, "p" )
                .up().add( AuditEntity.or( AuditEntity.property( "p", "age").eq( 20 ),
                    AuditEntity.relatedId( "owner" ).eq( null ) ) )
                .addOrder( AuditEntity.property( "make" ).asc() ).getResultList();
```

- 개정 번호 1에서는 제조가 "car3"인 모든 자동차와 소유자가 30세이거나 소유자가 없는 모든 자동차를 찾습니다.

```
List<Car> resultList = auditReader.createQuery()
                .forEntitiesAtRevision( Car.class, 1 )
                .traverseRelation( "owner", JoinType.LEFT, "p" )
                .up().add( AuditEntity.and( AuditEntity.property( "make" ).eq( "car3" ),
AuditEntity.property( "p", "age" ).eq( 30 ) ) )
                .getResultList();
```

- 개정 번호 1에서는 제조가 "car3"인 모든 자동차 또는 소유자가 10세 또는 소유자가 없는 경우를 찾습니다.

```
List<Car> resultList = auditReader.createQuery()
                .forEntitiesAtRevision( Car.class, 1 )
                .traverseRelation( "owner", JoinType.LEFT, "p" )
                .up().add( AuditEntity.or( AuditEntity.property( "make" ).eq( "car3" ),
AuditEntity.property( "p", "age" ).eq( 10 ) ) )
                .getResultList();
```

6.6. 성능 튜닝

6.6.1. 대체 배치 로드 알고리즘

Hibernate를 사용하면 join, select, subselect 및 batch의 네 가지 가져오기 전략 중 하나를 사용하여 연결에 대한 데이터를 로드할 수 있습니다. 이러한 네 가지 전략 중에서 배치 로드를 사용하면 선택 가져오기를 위한 최적화 전략이므로 가장 큰 성능상의 이점을 얻을 수 있습니다. 이 전략에서 Hibernate는 기본 또는 외부 키 목록을 지정하여 단일 SELECT 문으로 엔터티 인스턴스 또는 컬렉션 배치를 검색합니다. 배치 가져오기는 지연 선택 가져오기 전략의 최적화입니다.

배치 가져오기를 구성하는 방법은 클래스별 수준 또는 수집 수준입니다.

- 클래스별 수준
  Hibernate는 각 수준에서 데이터를 로드할 때 쿼리 시 사전 로드에 연관의 배치 크기가 필요합니다. 예를 들어 런타임 시 세션에 로드된 **Car** 오브젝트의 인스턴스 30개가 있다고 가정합니다. 각**Car** 오브젝트는 소유자 오브젝트에 속합니다. 모든자동차 객체를 반복하여 소유자를 요청하는 경우지연 로드를 통해 Hibernate는 각 소유자당 하나씩 30개의 선택 문을 발행합니다. 성능 병목 현상입니다.

  대신, Hibernate에 쿼리를 통해 검색되기 전에 다음 소유자 배치에 대한 데이터를 미리 로드하도록 지시할 수 있습니다. 소유자 오브젝트를 쿼리하면 Hibernate는 동일한 SELECT 문에서 이러한 오브젝트를 훨씬 더 쿼리합니다.

  사전에 쿼리할 소유자 오브젝트 수는 구성 시 지정된**batch-size** 매개변수에 따라 다릅니다.

  ```
  <class name="owner" batch-size="10"></class>
  ```

  이는 Hibernate에 가까운 미래에 필요할 것으로 예상하는 소유자 개체 10개를 더 쿼리하도록 지시합니다. 사용자가 자동차 **A** 의 소유자를 쿼리하면 **B** 의 소유자가 배치 로드의 일부로 이미 로드되었을 수 있습니다. 사용자가 실제로 자동차 **B** 의 소유자가 필요한 경우 데이터베이스로 이동하고 SELECT 문을 발행하는 대신 현재 세션에서 값을 검색할 수 있습니다.

  Hibernate 4.2.0은 **batch-size** 매개 변수 외에도 배치 로드 성능을 개선하기 위한 새 구성 항목을 도입했습니다. 구성 항목을 **Batch Fetchstyle** 구성이라고 하며 **hibernate.batch_fetch_style** 매개변수로 지정합니다.

  다음 세 가지 배치 가져오기 스타일이 지원됩니다. LEGACY, PADDED 및 DYNAMIC. 사용할 스타일을 지정하려면 **org.hibernate.cfg.AvailableSettings#BATCH_FETCH_STYLE** 을 사용합니다.

  - 레거시: 레거시 로드 스타일에서는 **ArrayHelper.getBatchSizes(int)** 를 기반으로 미리 구축된 배치 크기 세트를 사용합니다. 배치는 기존 배치 가능한 식별자 수에서 차세대 미리 빌드된 배치 크기를 사용하여 로드됩니다.
    위의 예제에서 배치 크기 설정이 **30**이면 미리 구성된 배치 크기는 [30, 15, 10, 9, 8, 7, .., 1]입니다. 부하 29개 식별자를 배치하려고 하면 15, 10, 4가 배치됩니다. 해당 SQL 쿼리에는 각각 데이터베이스에서 15, 10 및 4 소유자가 로드됩니다.

  - PADDED - Padded는 배치 로드의 LEGACY 스타일과 유사합니다. 여전히 미리 구축된 배치 크기를 활용하지만, 다음 큰 배치 크기를 사용하고 추가 식별자 자리 표시자를 패치합니다.
    위 예제와 마찬가지로 30개 소유자 오브젝트를 초기화해야 하는 경우 데이터베이스에 대해 하나의 쿼리만 실행됩니다.

    그러나 29개 소유자 개체를 초기화할 경우 Hibernate는 여전히 배치 크기 30의 SQL select 문 하나만 실행하고, 추가 공간은 반복 식별자로 채워집니다.

  - 동적 - 배치 크기 제한 사항을 준수하지만 이 배치 로드 스타일은 로드할 실제 오브젝트 수를 사용하여 SQL SELECT 문을 동적으로 빌드합니다.
    예를 들어 30개 소유자 개체의 경우 최대 배치 크기가 30이면 30개 소유자 개체를 검색하면 하나의 SQL SELECT 문이 생성됩니다. 35번 검색 호출을 실행하면 각각 배치 크기가 30 및 5인 두 개의 SQL 문이 생성됩니다. Hibernate는 배치 크기로 30이라는 제한 하에 유지되도록 두 번째 SQL 문을 동적으로 5개로 변경합니다. 두 번째 SQL은 PADDED를 가져오지 않으며 LEGACY 스타일과는 달리 두 번째 SQL 문에는 고정된 크기가 없습니다. 두 번째 SQL은 동적으로 생성됩니다.

    30개 이하의 식별자를 쿼리하는 경우 이 스타일은 요청된 식별자 수만 동적으로 로드합니다.

- 수집 기준 수준

Hibernate는 위의 클래스 섹션에 나열된 대로 배치 가져오기 크기 및 스타일을 이행하는 배치 로드 컬렉션도 수행할 수 있습니다.

이전 섹션에서 사용된 예제를 되돌리려면 각 소유자 오브젝트가 소유한 모든**Car** 오브젝트를 로드해야 한다고 고려합니다. 모든 소유자를 통해 반복되는 현재 세션에 10개의 소유자 개체가 로드되면 10개의 SELECT 문이 생성되며, 각각 **getœs()** 호출에 사용됩니다. 소유자 매핑에서 자동차 컬렉션에 대한 배치 가져오기를 활성화하면 Hibernate는 다음과 같이 이러한 컬렉션을 미리 가져올 수 있습니다.

```
<class name="Owner"><set name="cars" batch-size="5"></set></class>
```

따라서 배치 크기가 5개이고 레거시 배치 스타일을 사용하여 10개의 컬렉션을 로드하는 경우 Hibernate는 두 개의 SELECT 문으로 각각 5개의 컬렉션을 검색합니다.

6.6.2. 변경 불가능한 데이터에 대한 개체 참조의 두 번째 수준 캐싱

Hibernate는 성능 향상을 위해 메모리 내에서 자동으로 데이터를 캐시합니다. 이 작업은 특히 거의 변경되지 않는 데이터를 위해 데이터베이스 조회가 필요한 횟수를 줄여주는 인메모리 캐시를 통해 수행됩니다.

Hibernate는 두 가지 유형의 캐시를 유지 관리합니다. 첫 번째 수준 캐시라고도 하는 기본 캐시는 필수입니다. 이 캐시는 현재 세션과 연결되며 모든 요청이 이를 통과해야 합니다. 보조 캐시(두 번째 수준 캐시라고도 함)는 선택 사항이며 기본 캐시를 참조한 후에만 참조됩니다.

데이터는 먼저 상태 배열로 제거하여 두 번째 수준 캐시에 저장됩니다. 이 배열은 깊이 복사되며 해당 깊은 사본이 캐시에 배치됩니다. 그 반대로 캐시에서 읽을 수 있습니다. 이는 변경 가능한 데이터(변경 불가능한 데이터)에 적합하지만 변경 불가능한 데이터에는 비효율적입니다.

심도 있는 복사 데이터는 메모리 사용 및 처리 속도 측면에서 비용이 많이 드는 작업입니다. 대용량 데이터 세트의 경우 메모리 및 처리 속도가 성능 제한 요소가 됩니다. Hibernate를 사용하면 복사하지 않고 변경할 수 없는 데이터를 참조하도록 지정할 수 있습니다. 이제 전체 데이터 세트를 복사하는 대신 Hibernate는 이제 캐시의 데이터에 대한 참조를 저장할 수 있습니다.

구성 설정 **hibernate.cache.use_reference_entries** 값을 **true** 로 변경하여 수행할 수 있습니다. 기본적으로 **hibernate.cache.use_reference_entries** 는 **false** 로 설정됩니다.

hibernate.cache.use_reference_entries 가 **true** 로 설정되면 연관이 없는 변경 불가능한 데이터 오브젝트가 두 번째 수준 캐시에 복사되지 않고 해당 오브젝트에 대한 참조만 저장됩니다.

주의

hibernate.cache.use_reference_entries 를 **true** 로 설정하면 연관이 있는 변경 불가능한 데이터 오브젝트가 여전히 두 번째 수준 캐시에 복사됩니다.

7장. HIBERNATE SEARCH

7.1. HIBERNATE SEARCH 시작하기

7.1.1. Hibernate Search 정보

Hibernate Search는 Hibernate 애플리케이션에 전체 텍스트 검색 기능을 제공합니다. 전체 텍스트, 퍼지 및 지리적 위치 검색을 포함하여 SQL 기반 솔루션이 적합하지 않은 애플리케이션을 검색하는 데 특히 적합합니다. Hibernate Search는 Apache Lucene을 전체 텍스트 검색 엔진으로 사용하지만 유지 관리 오버헤드를 최소화하도록 설계되었습니다. 구성되고 나면 인덱싱, 클러스터링 및 데이터 동기화가 투명하게 유지되므로 비즈니스 요구 사항을 충족하는 데 집중할 수 있습니다.

참고

이전 JBoss EAP 릴리스에는 Hibernate 4.2 및 Hibernate Search 4.6이 포함되었습니다. JBoss EAP 7에는 Hibernate 5 및 Hibernate Search 5.5가 포함되어 있습니다.

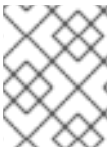
네이티브 Lucene API를 사용하는 경우 이 버전에 맞게 조정하십시오.

7.1.2. Hibernate 검색 개요

Hibernate Search는 인덱스 검색 구성 요소뿐만 아니라 색인 검색 구성 요소로 구성되어 있으며 둘 다 Apache Lucene에서 지원합니다. 데이터베이스에서 엔터티를 삽입, 업데이트 또는 제거할 때마다 Hibernate Search는 Hibernate 이벤트 시스템을 통해 이 이벤트를 추적하고 인덱스 업데이트를 예약합니다. 이러한 모든 업데이트는 Apache Lucene API와 직접 상호 작용할 필요 없이 처리됩니다. 대신 기본 Lucene 인덱스와의 상호 작용은 **IndexManager** 를 통해 처리됩니다. 기본적으로 IndexManager와 Lucene 인덱스 사이에는 일대일 관계가 있습니다. IndexManager는 선택한 *백엔드*,*reader 전략*및 *DirectoryProvider* 를 포함한 특정 인덱스 구성을 추상화합니다.

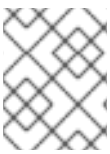
인덱스가 생성되면 기본 Lucene 인프라를 처리하는 대신 엔터티를 검색하고 관리 엔터티 목록을 반환할 수 있습니다. Hibernate와 Hibernate Search 간에 동일한 지속성 컨텍스트를 공유합니다. **FullTextSession** 클래스는 Hibernate 세션 클래스 위에 구축되어 애플리케이션 코드가 통합**org.hibernate.Query** 또는 **javax.persistence.Query** API와 동일한 방식으로 HQL, Jakarta Persistence 쿼리 언어 또는 네이티브 쿼리와 정확히 동일합니다.

트랜잭션 배치 모드는 Java 네이밍 및 디렉터리 인터페이스를 기반으로 하는지 여부에 관계없이 모든 작업에 권장됩니다.



참고

Java Naming 및 Directory Interface 또는 Jakarta Transactions와 관계없이 데이터베이스와 Hibernate Search 모두 트랜잭션에서 작업을 실행하는 것이 좋습니다.



참고

Hibernate Search는 원자성 대화로 알려진 Hibernate 또는 EntityManager 긴 대화 패턴에서 완벽하게 작동합니다.

7.1.3. 디렉터리 공급자 정보

Hibernate Search 인프라의 일부인 Apache Lucene은 인덱스 저장 디렉터리의 개념을 가지고 있습니다. Hibernate Search는 디렉터리 *공급자*를 통해 Lucene Directory 인스턴스의 초기화 및 구성을 처리합니다.

directory_provider 속성은 인덱스를 저장하는 데 사용할 디렉터리 공급자를 지정합니다. 기본 파일 시스템 디렉터리 프로바이더는 로컬 파일 시스템을 사용하여 인덱스를 저장하는 **filesystem** 입니다.

7.1.4. 작업자 정보

Lucene 인덱스 업데이트는 Hibernate Search *Worker* 에서 처리합니다. 이 작업자는 모든 엔터티 변경 사항을 수신하고 컨텍스트별로 대기열을 지정한 다음 컨텍스트가 종료되면 적용합니다. 가장 일반적인 컨텍스트는 트랜잭션이지만 엔터티 변경 수 또는 기타 애플리케이션 이벤트 수에 따라 달라질 수 있습니다.

효율성을 높이기 위해 상호 작용은 일괄 처리되며 컨텍스트가 끝나면 일반적으로 적용됩니다. 트랜잭션 외부에서는 실제 데이터베이스 작업 후에 인덱스 업데이트 작업이 실행됩니다. 진행 중인 트랜잭션의 경우 인덱스 업데이트 작업은 트랜잭션 커밋 단계에 맞게 예약되고 트랜잭션 롤백 시 삭제됩니다. 작업자는 컨텍스트에 관계없이 인덱싱이 수행된 후 특정 배치 크기 제한을 사용하여 구성할 수 있습니다.

이 인덱스 업데이트를 처리하는 방법에는 두 가지 즉각적인 이점이 있습니다.

- 성능: 배치에서 작업을 실행할 때 Lucene 인덱싱이 더 효과적입니다.
- 초대도: 실행된 작업은 데이터베이스 트랜잭션에서 실행하는 것과 동일한 범위를 가지며 트랜잭션이 커밋된 경우에만 실행됩니다. 이는 엄격한 의미에서는 ACID가 아니지만, 언제든지 소스에서 다시 빌드할 수 있으므로 ACID 동작은 전체 텍스트 검색 색인에 거의 유용하지 않습니다.

범위와 트랜잭션이 없는 두 개의 배치 모드는 자동 커밋과 트랜잭션 동작에 해당합니다. 성능 관점에서 *트랜잭션* 모드가 권장됩니다. 범위 지정 선택은 투명하게 이루어집니다. Hibernate Search는 트랜잭션의 존재를 감지하고 범위를 조정합니다.

7.1.5. 백엔드 설정 및 작업

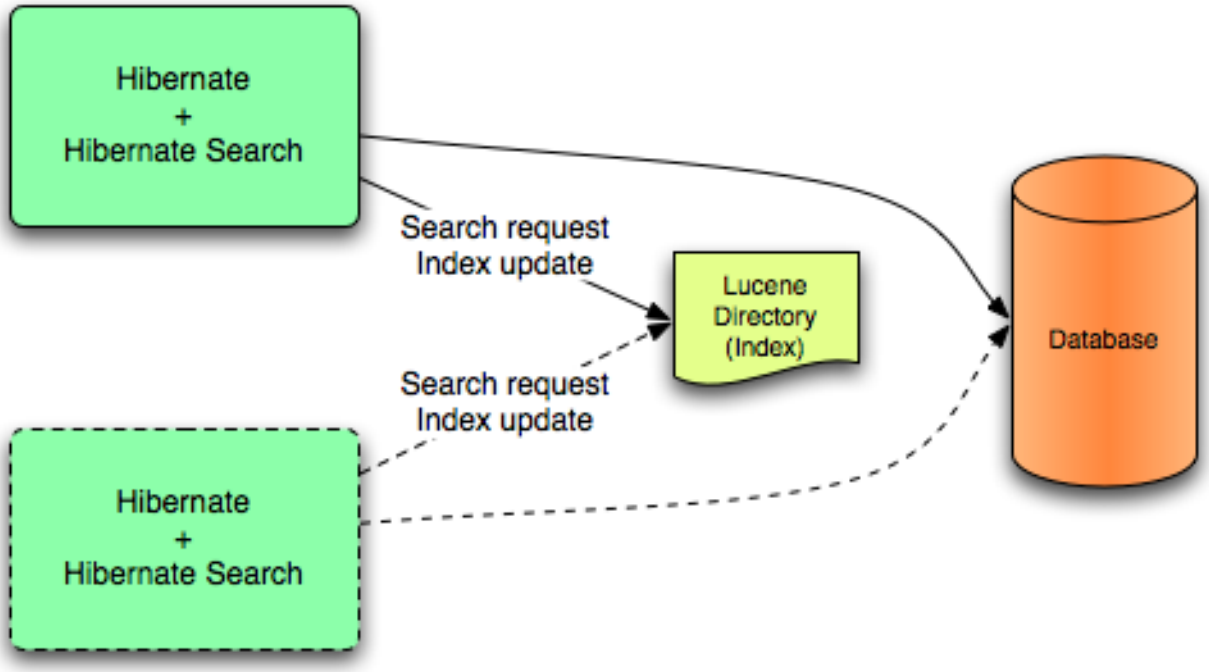
7.1.5.1. 백엔드

Hibernate Search는 다양한 백엔드를 사용하여 작업 배치를 처리합니다. 백엔드는 구성 옵션 **default.worker.backend** 로 국한되지 않습니다. 이 속성은 백엔드 구성의 일부인백엔드**QueueProcessor** 인터페이스의 구현을 지정합니다. 백엔드(예: Jakarta Messaging 백엔드)를 설정하려면 추가 설정이 필요합니다.

7.1.5.2. Lucene

Lucene 모드에서는 노드의 모든 인덱스 업데이트가 디렉터리 공급업체를 사용하여 Lucene 디렉터리에 동일한 노드에서 실행됩니다. 비클러스터형 환경이나 공유 디렉터리 저장소가 있는 클러스터형 환경에서 이 모드를 사용합니다.

그림 7.1. Lucene 백엔드 구성

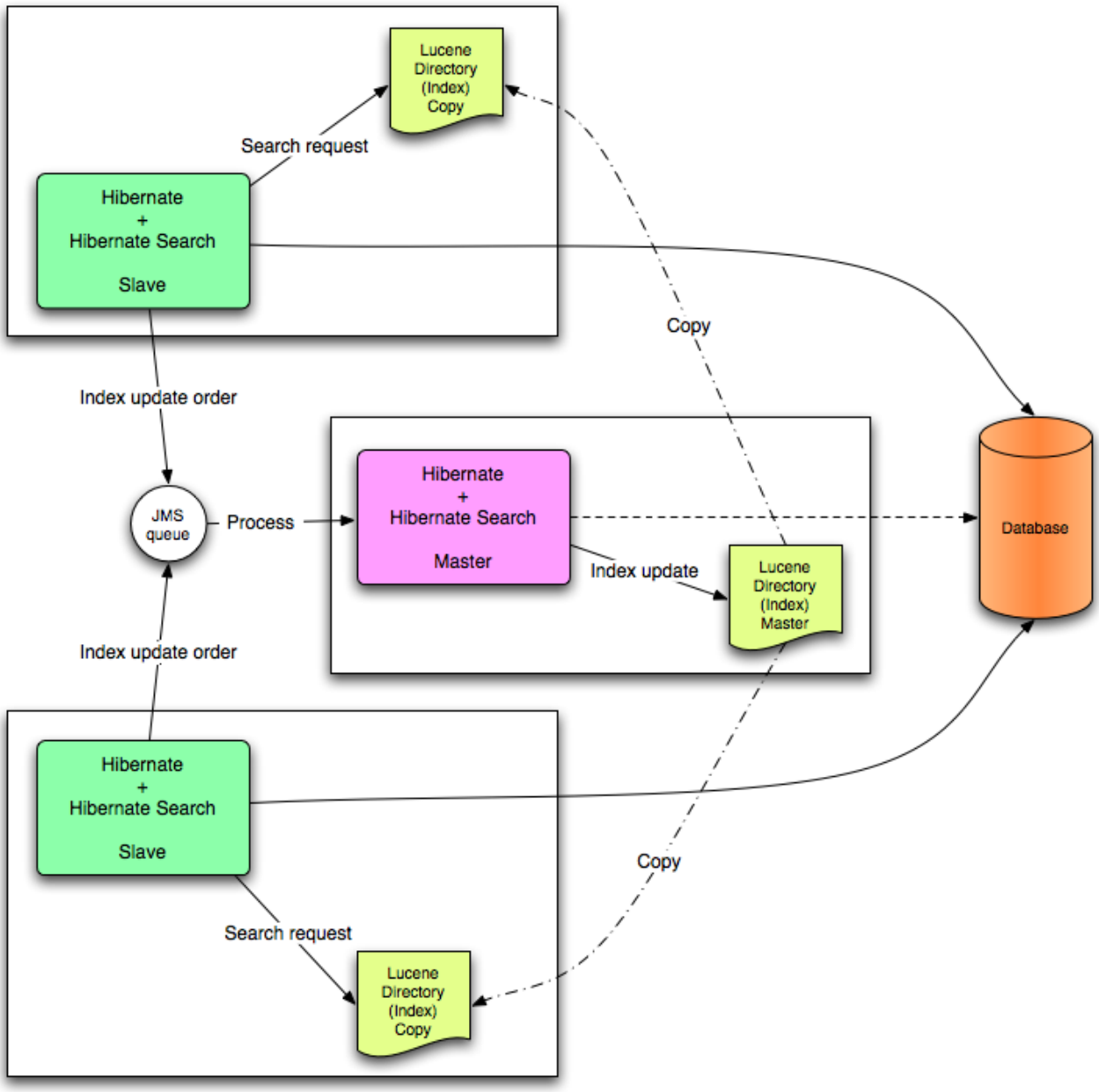


Lucene 모드는 디렉터리가 잠금 전략을 관리하는 비클러스터형 또는 클러스터된 애플리케이션을 대상으로 합니다. Lucene 모드의 주요 장점은 단순하고 Lucene 쿼리의 변경 사항을 즉시 확인하는 것입니다. NRT(Non-Real Time) 백엔드는 비클러스터형 인덱스 구성 및 비공유 인덱스 구성의 대체 백엔드입니다.

7.1.5.3. 자카르타 메시징

노드의 인덱스 업데이트가 자카르타 메시징 큐로 전송됩니다. 고유한 리더는 대기열을 처리하고 마스터 인덱스를 업데이트합니다. 이후 마스터 인덱스는 마스터 및 슬레이브 패턴을 설정하기 위해 슬레이브 복사본에 정기적으로 복제됩니다. 마스터는 Lucene 인덱스 업데이트를 담당합니다. 슬레이브는 읽기 및 쓰기 작업을 허용하지만 로컬 인덱스 복사본에 대한 읽기 작업을 처리합니다. 마스터는 Lucene 인덱스를 업데이트하는 데만 책임이 있습니다. 마스터만 업데이트 작업에 로컬 변경 사항을 적용합니다.

그림 7.2. 자카르타 메시징 서비스 백엔드 구성



이 모드는 처리량이 중요하고 인덱스 업데이트 지연이 경제적인 클러스터형 환경을 대상으로 합니다. 자카르타 메시징 공급자는 안정성을 보장하고 슬레이브를 사용하여 로컬 인덱스 복사본을 변경합니다.

7.1.6. 독자 전략

쿼리를 실행할 때 Hibernate Search는 reader 전략을 사용하여 Apache Lucene 인덱스와 상호 작용합니다. 자주 사용하는 업데이트와 같은 애플리케이션의 프로필을 기반으로 reader 전략을 선택하고, 주로 읽기, 비동기 인덱스 업데이트.

7.1.6.1. 공유 전략

Hibernate Search는 공유 전략을 사용하여 **IndexReader** 가 계속해서 업데이트되는 제공된 여러 쿼리와 스레드에서 주어진 Lucene 인덱스에 대해 동일한 **IndexReader** 를 공유합니다. **IndexReader** 가 업데이트되지 않으면 새 항목이 열리고 제공됩니다. 각 **IndexReader** 는 여러 **SegmentReaders** 로 구성됩니다. 공유 전략은 마지막 열기 후에 수정되거나 생성된 세그먼트를 다시 열고 이전 인스턴스에서 이미 로드된 세그먼트를 공유합니다. 기본 전략입니다.

7.1.6.2. 비공유 전략

비공유 전략을 사용하면 쿼리가 실행될 때마다 Lucene**IndexReader** 가 열립니다. **IndexReader** 열기 및 시작은 비용이 많이 드는 작업입니다. 따라서 각 쿼리 실행에 대해 **IndexReader** 를 여는 것은 효율적인 전략이 아닙니다.

7.1.6.3. 사용자 정의 리더 전략

org.hibernate.search.reader.ReaderProvider 구현을 사용하여 사용자 정의 리더 전략을 작성할 수 있습니다. 구현은 안전한 스레드여야 합니다.

7.2. 설정

7.2.1. 최소 설정

Hibernate Search는 구성 및 작업에 유연성을 제공하도록 설계되었으며, 대부분의 사용 사례에 맞게 신중하게 선택한 기본값을 사용합니다. 최소한 디렉터리 공급업체는 속성과 함께 구성해야 합니다. 기본 디렉터리 프로바이더는 인덱스 스토리지에 로컬 파일 시스템을 사용하는 **filesystem** 입니다. 사용 가능한 디렉터리 공급자 및 해당 구성에 대한 자세한 내용은 DirectoryProvider Configuration을 참조하십시오.

Hibernate를 직접 사용하는 경우 DirectoryProvider와 같은 설정은 구성 파일에서 **hibernate.properties** 또는 **hibernate.cfg. xml** 중 하나를 설정해야 합니다. Jakarta Persistence를 통해 Hibernate를 사용하는 경우 구성 파일은 **persistence.xml** 입니다.

추가 리소스

- 사용 가능한 디렉터리 공급자 및 해당 구성에 대한 자세한 내용은 DirectoryProvider Configuration을 참조하십시오.

7.2.2. IndexManager 구성

Hibernate Search는 이 인터페이스에 대해 몇 가지 구현을 제공합니다.

- 디렉터리 기반: Lucene **Directory** 추상화를 사용하여 인덱스 파일을 관리하는 기본 구현입니다.
- **near-live-time**: 각 커밋 시 디스크에 쓰기를 플러시하지 않도록 합니다. 또한 이 인덱스 관리자는디렉터리 기반이지만 Lucene의 거의 실시간 NRT 기능을 사용합니다.

기본값 이외의 IndexManager를 지정하려면 다음 속성을 지정합니다.

```
hibernate.search.[default|<indexname>].indexmanager = near-real-time
```

7.2.2.1. 디렉토리 기반

디렉터리 기반 구현은 기본 **IndexManager** 구현입니다. 구성 가능하며 reader 전략, 백엔드 및 디렉터리 공급업체에 대해 별도의 구성을 허용합니다.

7.2.2.2. 거의 실시간

NRTIndexManager 는 기본 **IndexManager** 의 확장이며 짧은 대기 시간 색인 쓰기 기능을 위한 Lucene NRT, near Real Time을 활용합니다. 하지만 **lucene** 이 아닌 대체 백엔드에 대한 구성 설정을 무시하고 **Directory** 에서 독점적인 쓰기 잠금을 갖게 됩니다.

IndexWriter 는 짧은 대기 시간을 제공하기 위해 디스크의 모든 변경을 플러시하지 않습니다. 쿼리는 플러시되지 않은 인덱스 작성기 버퍼에서 업데이트된 상태를 읽을 수 있습니다. 그러나 **IndexWriter** 가 종료되거나 애플리케이션 충돌이 발생하면 업데이트가 손실되어 인덱스를 다시 빌드해야 합니다.

언급한 단점으로 인해 데이터가 제한된 클러스터되지 않은 웹 사이트에는 실시간 구성이 권장되며 성능 향상을 위해 마스터 노드를 개별적으로 구성할 수 있기 때문입니다.

7.2.2.3. 사용자 지정

사용자 정의 구현에 대해 정규화된 클래스 이름을 지정하여 사용자 지정 **IndexManager** 를 설정합니다. 다음과 같이 구현에 대한 인수 없는 생성자를 설정합니다.

```
[default|<indexname>].indexmanager = my.corp.myapp.CustomIndexManager
```

사용자 정의 인덱스 관리자 구현에는 기본 구현과 동일한 구성 요소가 필요하지 않습니다. 예를 들어 디렉터리 인터페이스를 노출하지 않는 원격 인덱싱 서비스에 위임합니다.

7.2.3. DirectoryProvider 설정

DirectoryProvider 는 Lucene **Directory** 를 중심으로 Hibernate Search 추상화이며 기본 Lucene 리소스의 구성 및 초기화를 처리합니다. 디렉터리 공급자와 해당 속성은 Hibernate Search에서 사용할 수 있는 디렉터리 공급자 목록과 해당 옵션을 표시합니다.

인덱스가 지정된 각 엔터티는 Lucene 인덱스와 연결됩니다(여러 엔터티가 동일한 인덱스를 공유하는 경우 제외). 인덱스 이름은 **@Indexed** 주석의 **index** 속성에서 제공합니다. 인덱스 속성을 지정하지 않으면 색인화된 클래스의 정규화된 이름이 이름(권장됨)으로 사용됩니다.

DirectoryProvider 및 추가 옵션은 접두사 **hibernate.search.*<indexname>*** 을 사용하여 구성할 수 있습니다. **default** 라는 이름(**hibernate.search.default**)은 예약되어 있으며 모든 인덱스에 적용되는 속성을 정의하는 데 사용할 수 있습니다. 디렉터리 프로바이더 구성은 **hibernate.search.default.directory_provider** 를 사용하여 기본 디렉터리 프로바이더를 파일 시스템으로 설정하는 방법을 보여줍니다. **hibernate.search.default.indexBase** 다음에 인덱스의 기본 기본 디렉터리가 설정됩니다. 결과적으로 엔터티 **Status** 의 인덱스가 **/usr/lucene/indexes/org.hibernate.example.Status** 에 생성됩니다.

그러나 이 엔터티의 기본 디렉터리 프로바이더가 **hibernate.search.Rules.directory_provider** 속성으로 재정의되므로 **Rule** 엔터티에 대한 인덱스는 메모리 내 디렉터리를 사용하고 있습니다.

마지막으로 **Action** 엔터티는 **hibernate.search.Actions.directory_provider** 를 통해 지정된 사용자 지정 디렉터리 프로바이더 **CustomDirectoryProvider** 를 사용합니다.

인덱스 이름 지정

```
package org.hibernate.example;

@Indexed
public class Status { ... }

@Indexed(index="Rules")
public class Rule { ... }

@Indexed(index="Actions")
public class Action { ... }
```

디렉터리 공급자 구성

```
hibernate.search.default.directory_provider = filesystem
hibernate.search.default.indexBase=/usr/lucene/indexes
hibernate.search.Rules.directory_provider = ram
hibernate.search.Actions.directory_provider = com.acme.hibernate.CustomDirectoryProvider
```

참고

설명된 구성 체계를 사용하면 디렉터리 프로바이더 및 기본 디렉터리와 같은 일반적인 규칙을 쉽게 정의하고 인덱스별로 나중에 기본값을 재정의할 수 있습니다.

디렉토리 공급자 및 해당 속성

RAM

없음

파일 시스템

파일 시스템 기반 디렉토리. 사용되는 디렉토리는 <indexBase>/< indexName >입니다.

- indexBase : 기본 디렉토리
- indexName: @Indexed.index 재정의 (sharded indexes에 유용)
- locking_strategy : optional, see LockFactory Configuration
- filesystem_access_type: 이 **DirectoryProvider** 에서 사용하는 **FSDirectory** 구현의 정확한 유형을 확인할 수 있습니다. 허용되는 값은 **auto** (기본값, Windows 시스템의 **NIOFSDirectory**, Windows의 **SimpleFSDirectory**), **simple(SimpleFSDirectory), Nio(NIOFSDirectory), mmap (MMapDirectory)** 을 선택합니다. 이 설정을 변경하기 전에 이러한 디렉터리 구현에 대한 Javadoc를 참조하십시오. **NIOFSDirectory** 또는 **MMapDirectory** 는 상당한 성능 향상을 가져올 수 있지만 문제도 있습니다.

filesystem-master

파일 시스템 기반 디렉토리. 파일 시스템 처럼. 또한 정기적으로 인덱스를 소스 디렉터리(복사 디렉터리라고 함)에 복사합니다.
새로 고침 기간에 권장되는 값은 정보를 복사하는 시간(기본값: 3600초 - 60분)의 50% 더 높습니다.

복사본은 평균 복사 시간을 줄이는 증분 복사 메커니즘을 기반으로 합니다.

DirectoryProvider는 일반적으로 자카르타 메시징 백엔드 클러스터의 마스터 노드에서 사용됩니다.

buffer_size_on_copy 최적의 크기는 운영 체제와 사용 가능한 RAM에 따라 다릅니다. 대부분의 사람들은 16MB에서 64MB 사이의 값을 사용하여 좋은 결과를 보고했습니다.

- indexBase: 기본 디렉토리
- indexName: @Indexed.index 재정의 (sharded indexes에 유용)
- sourceBase: 소스 (복사) 기본 디렉토리.
- Source: 소스 디렉토리 접미사 (기본값: **@Indexed.index**). 실제 소스 디렉토리 이름이 **<sourceBase>/<source>**입니다.
- refresh: 새로 고침 기간(초)입니다(복제는 새로 고침 초마다 수행됨). 다음새로 고침 기간이 경과할 때 복사본이 계속 진행 중인 경우 두 번째 복사 작업을 건너뜁니다.

- buffer_size_on_copy: 단일 낮은 수준의 복사 명령으로 이동할 바이트 양입니다. 기본값은 16MB입니다.
- locking_strategy : optional, see LockFactory Configuration
- filesystem_access_type: 이 **DirectoryProvider** 에서 사용하는 **FSDirectory** 구현의 정확한 유형을 확인할 수 있습니다. 허용되는 값은 **auto** (기본값, Windows 시스템의 **NIOFSDirectory**, Windows의 **SimpleFSDirectory**), **simple(SimpleFSDirectory)**, **Nio(NIOFSDirectory)**, **mmap (MMapDirectory)** 을 선택합니다. 이 설정을 변경하기 전에 이러한 디렉터리 구현에 대한 Javadoc를 참조하십시오. **NIOFSDirectory** 또는 **MMapDirectory** 는 상당한 성능 향상을 가져올 수 있지만, 인식해야 하는 문제도 있습니다.

filesystem-slave

파일 시스템 기반 디렉토리. 파일 시스템과 유사하지만 정기적으로 마스터 버전(소스)을 검색합니다. 잠금 및 일관성 없는 검색 결과를 방지하기 위해 로컬 복사본 2개가 유지됩니다.
새로 고침 기간에 권장되는 값은 정보를 복사하는 시간(기본값: 3600초 - 60분)의 50% 더 높습니다.

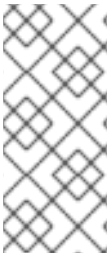
복사본은 평균 복사 시간을 줄이는 증분 복사 메커니즘을 기반으로 합니다. 새로 고침 기간이 경과할 때 복사본이 계속 진행 중이면 두 번째 복사 작업을 건너뜁니다.

DirectoryProvider는 일반적으로 자카르타 메시징 백엔드를 사용하는 슬레이브 노드에서 사용됩니다.

buffer_size_on_copy 최적의 크기는 운영 체제와 사용 가능한 RAM에 따라 다릅니다. 대부분의 사람들은 16MB에서 64MB 사이의 값을 사용하여 좋은 결과를 보고했습니다.

- indexBase: 기본 디렉터리
- indexName: @Indexed.index 재정의 (sharded indexes에 유용)
- sourceBase: 소스(복사) 기본 디렉터리.
- source: 소스 디렉터리 접미사(기본값: **@Indexed.index**). 실제 소스 디렉토리 이름이 **<sourceBase>/<source>**입니다.
- refresh: 새로 고침 간격(초)입니다(복제는 새로 고침 초마다 수행됨).
- buffer_size_on_copy: 단일 낮은 수준의 복사 명령으로 이동할 바이트 양입니다. 기본값은 16MB입니다.
- locking_strategy : optional, see LockFactory Configuration
- retry_marker_lookup : 선택 사항, 기본값은 0입니다. 실패하기 전에 Hibernate Search에서 소스 디렉터리에서 마커 파일을 검사하는 횟수를 정의합니다. 각 시도 사이에 5초 동안 기다립니다.
- retry_initialize_period : 선택 사항으로 재시도 초기화 기능을 사용하도록 정수 값을 초 단위로 설정합니다. 슬레이브가 마스터 인덱스를 찾을 수 없는 경우 애플리케이션이 시작되지 않도록 백그라운드에서 찾을 때까지 다시 시도됩니다. 인덱스를 초기화하기 전에 수행된 전체 Text 쿼리는 차단되지 않지만 빈 결과를 반환합니다. 옵션을 활성화하거나 명시적으로 0으로 설정하지 않으면 재시도 타이머를 예약하는 대신 예외적으로 실패합니다. 애플리케이션이 유효하지 않은 인덱스 없이 시작되지 않지만 초기화 시간 초과를 계속 제어하려면 대신 **retry_marker_lookup** 을 참조하십시오.
- filesystem_access_type: 이 **DirectoryProvider** 에서 사용하는 **FSDirectory** 구현의 정확한 유형을 확인할 수 있습니다. 허용되는 값은 auto(기본값, Windows 시스템의 **NIOFSDirectory**, Windows의 **SimpleFSDirectory**), **simple(SimpleFSDirectory)**, **Nio(NIOFSDirectory)**,

mmap (MMapDirectory) 을 선택합니다. 이 설정을 변경하기 전에 이러한 디렉터리 구현에 대한 Javadoc를 참조하십시오. **NIOFSDirectory** 또는 **MMapDirectory** 는 상당한 성능 향상을 가져올 수 있지만 문제를 알아야 합니다.



참고

기본 제공 디렉터리 공급자가 요구 사항에 맞지 않는 경우 **org.hibernate.store.DirectoryProvider** 인터페이스를 구현하여 자체 디렉터리 공급자를 작성할 수 있습니다. 이 경우 공급자의 정규화된 클래스 이름을 **directory_provider** 속성에 전달합니다. 접두사 **hibernate.search.*<indexname>*** 을 사용하여 추가 속성을 전달할 수 있습니다.

7.2.4. 작업자 구성

Hibernate Search가 작업자 구성을 통해 Lucene과 상호 작용하는 방식을 구체화할 수 있습니다. 여러 아키텍처 구성 요소와 가능한 확장 지점이 있습니다. 좀 더 자세히 살펴보겠습니다.

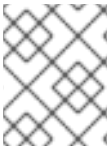
작업자 구성을 사용하여 Infinispan 쿼리가 Lucene과 상호 작용하는 방식을 구체화합니다. 이 구성에 여러 아키텍처 구성 요소와 가능한 확장 지점을 사용할 수 있습니다.

먼저 작업자가 있습니다. **Worker** 인터페이스 구현은 모든 엔터티 변경 사항을 수신하여 컨텍스트를 통해 대기열에 추가한 후 컨텍스트가 종료되면 적용하는 작업을 담당합니다. 특히 ORM과 관련하여 가장 직관적인 컨텍스트는 트랜잭션입니다. 이러한 이유로 Hibernate Search는 기본적으로 **TransactionalWorker** 를 사용하여 트랜잭션당 모든 변경 사항을 범위를 지정합니다. 그러나 컨텍스트가 엔터티 변경 수나 기타 애플리케이션 라이프사이클 이벤트 등에 따라 달라지는 시나리오가 있다고 가정할 수 있습니다.

표 7.1. 범위 설정

속성	설명
hibernate.search.worker.scope	사용할 **Worker** 구현의 정규화된 클래스 이름입니다. 이 속성이 설정되지 않은 경우 비어 있거나 **트랜잭션** 하는 경우 기본 **TransactionalWorker** 가 사용됩니다.
hibernate.search.worker.*	**hibernate.search.worker** 접두사가 지정된 모든 구성 속성은 초기화 중에 Worker로 전달됩니다. 이를 통해 사용자 지정 작업자별 매개변수를 추가할 수 있습니다.
hibernate.search.worker.batch_size	컨텍스트당 배치된 최대 인덱싱 작업 수를 정의합니다. 컨텍스트가 아직 종료되지 않았더라도 제한에 도달하면 인덱싱이 트리거됩니다. 이 속성은 **작업자** 구현이 **트랜잭션 작업인 BatchedQueueingProcessor 에 위임하는 경우에만 작동합니다.**

컨텍스트가 종료되면 인덱스 변경 사항을 준비하고 적용해야 합니다. 이 작업은 새 스레드 내에서 동기적으로 또는 비동기적으로 수행할 수 있습니다. 동기 업데이트에는 데이터베이스와 항상 인덱스가 동기화되는 이점이 있습니다. 반면 비동기 업데이트는 사용자 응답 시간을 최소화하는 데 도움이 될 수 있습니다. 단점은 데이터베이스와 인덱스 상태 간의 잠재적인 불일치입니다.



참고

다음 옵션은 인덱스마다 다를 수 있습니다. 실제로는 indexName 접두사가 필요하거나 default 를 사용하여 모든 인덱스의 기본값을 설정합니다.

표 7.2. 실행 설정

속성	설명
hibernate.search.<indexName>.worker.execution	**sync**: 동기 실행 (기본값) **async**: 비동기 실행
hibernate.search.<indexName>.worker.thread_pool.size	백엔드는 스레드 풀을 사용하여 동일한 트랜잭션 컨텍스트(또는 배치)의 업데이트를 병렬로 적용할 수 있습니다. 기본값은 1입니다. 트랜잭션당 작업이 많은 경우 더 큰 값으로 실험할 수 있습니다.
hibernate.search.<indexName>.worker.buffer_queue.max	스레드 풀이 종료된 경우 최대 작업 대기열 수를 정의합니다. 비동기 실행에만 유용합니다. 기본값은 무한입니다. 제한에 도달하면 기본 스레드에서 작업을 수행합니다.

지금까지 실행 모드에 관계없이 동일한 VM(가상 시스템) 내에서 모든 작업이 수행됩니다. 단일 VM에 대한 총 작업 양이 변경되지 않았습니다. 다행히 더 나은 접근 방식, 즉 위임이 있습니다. hibernate.search.default.worker.backend 를 구성하여 인덱싱 작업을 다른 서버로 보낼 수 있습니다. 다시 각 인덱스에 대해 이 옵션을 다르게 구성할 수 있습니다.

표 7.3. 백엔드 구성

속성	설명
hibernate.search.<indexName>.worker.backend	**Lucene**: 동일한 VM에서 인덱스 업데이트를 실행하는 기본 백엔드입니다. 속성이 정의되지 않았거나 비어 있는 경우에도 사용됩니다. **jms**: 자카르타 메시징 백엔드. 인덱스 업데이트는 인덱싱 마스터에서 처리하도록 자카르타 메시징 큐로 전송됩니다. 추가 구성 옵션 및 이 설정에 대한 자세한 설명은 Jakarta Messaging 백엔드 구성을 참조하십시오. **blackhole**: 주로 모든 인덱싱 작업을 무시하는 test/developer 설정 **백엔드QueueProcessor** 를 구현하는 클래스의 정규화된 이름을 지정할 수도 있습니다. 이렇게 하면 고유한 통신 계층을 구현할 수 있습니다. 구현은 실행 시 인덱스 작업을 처리하는 **Runnable** 인스턴스를 반환합니다.

1. **자카르타 메시징 백엔드 구성**

속성	설명
hibernate.search.<indexName>.worker.jndi.*	필요한 경우 InitialContext를 시작할 Java Naming 및 Directory Interface 속성을 정의합니다. Java 네이밍 및 디렉터리 인터페이스는 자카르타 메시징 백엔드에서만 사용합니다.
hibernate.search.<indexName>.worker.jms.connection_factory	자카르타 메시징 백엔드에는 필수 항목입니다. Java Naming 및 Directory Interface 이름을 정의하여 Jakarta Messaging 연결 팩토리를 조회합니다(**Red Hat JBoss Enterprise Application Platform에서 기본적으로 /ConnectionFactory**).
hibernate.search.<indexName>.worker.jms.queue	자카르타 메시징 백엔드에는 필수 항목입니다. 에서 자카르타 메시징 대기열을 조회할 Java Naming 및 Directory Interface 이름을 정의합니다. 작업 메시지를 게시하는 데 큐가 사용됩니다.

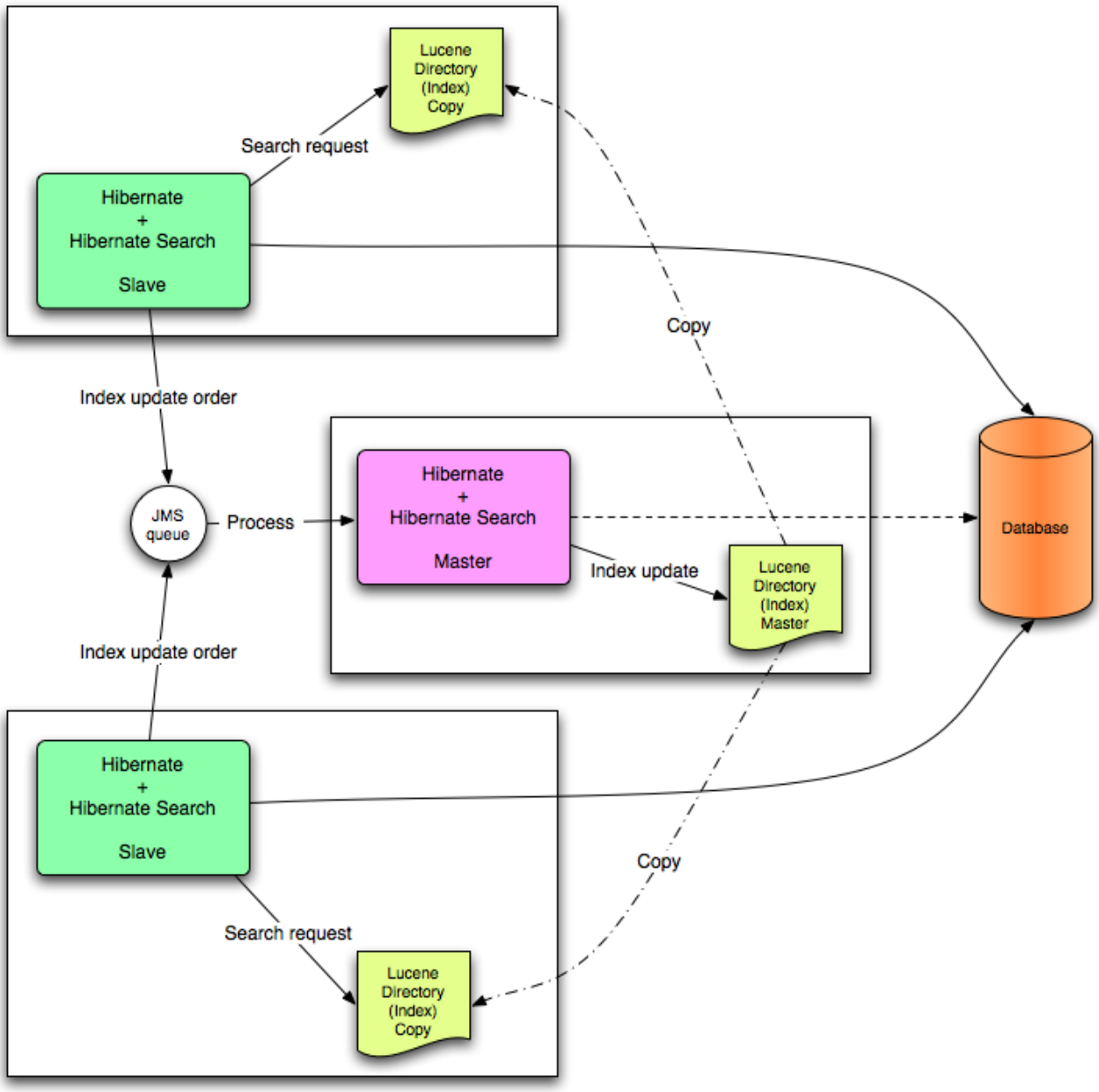
주의

분명한 바와 같이, 표시된 속성 중 일부는 상관 관계가 있으며 이는 속성 값의 모든 조합이 의미하지는 않음을 의미합니다. 실제로는 작동하지 않는 구성을 사용할 수 있습니다. 이는 표시된 인터페이스 중 일부에 대해 자체 구현을 제공하는 경우에 특히 해당합니다. 자체 Worker 또는 BackendQueueProcessor 구현을 작성하기 전에 기존 코드를 사용해야 합니다.

7.2.4.1. 자카르타 메시징 마스터/슬레이브 백엔드

이 섹션에서는 마스터/슬레이브 Hibernate Search 아키텍처를 구성하는 방법을 자세히 설명합니다.

그림 7.3. 자카르타 메시징 백엔드 구성



7.2.4.2. 슬레이브 노드

모든 인덱스 업데이트 작업은 자카르타 메시징 큐로 전송됩니다. 인덱스 쿼리 작업은 로컬 인덱스 사본에서 실행됩니다.

자카르타 메시징 슬래브 구성

```
### slave configuration

## DirectoryProvider
# (remote) master location
hibernate.search.default.sourceBase = /mnt/mastervolume/lucenedirs/mastercopy

# local copy location
hibernate.search.default.indexBase = /Users/prod/lucenedirs

# refresh every half hour
```

```
hibernate.search.default.refresh = 1800

# appropriate directory provider
hibernate.search.default.directory_provider = filesystem-slave

## Back-end configuration
hibernate.search.default.worker.backend = jms
hibernate.search.default.worker.jms.connection_factory = /ConnectionFactory
hibernate.search.default.worker.jms.queue = queue/hibernatesearch
#optional jndi configuration (check your Jakarta Messaging provider for more information)

## Optional asynchronous execution strategy
# hibernate.search.default.worker.execution = async
# hibernate.search.default.worker.thread_pool.size = 2
# hibernate.search.default.worker.buffer_queue.max = 50
```

참고

더 빠른 검색 결과를 얻으려면 파일 시스템 로컬 사본을 사용하는 것이 좋습니다.

7.2.4.3. 마스터 노드

모든 인덱스 업데이트 작업은 자카르타 메시징 큐에서 가져온 후 실행됩니다. 마스터 인덱스는 정기적으로 복사됩니다.

Jakarta Messaging 큐의 인덱스 업데이트 작업이 실행되고 마스터 인덱스가 정기적으로 복사됩니다.

자카르타 메시징 서비스 마스터 구성

```
### master configuration

## DirectoryProvider
# (remote) master location where information is copied to
hibernate.search.default.sourceBase = /mnt/mastervolume/lucenedirs/mastercopy

# local master location
hibernate.search.default.indexBase = /Users/prod/lucenedirs

# refresh every half hour
hibernate.search.default.refresh = 1800

# appropriate directory provider
hibernate.search.default.directory_provider = filesystem-master

## Back-end configuration
#Back-end is the default for Lucene
```

Hibernate Search 프레임워크 구성 외에도 메시지 기반 빈을 작성하고 자카르타 메시징을 통해 인덱스 작동 대기열을 처리하도록 설정해야 합니다.

메시지 기반 빈 인덱싱 대기열 처리

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName="destinationType",
```

```
                    propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="destination",
                    propertyValue="queue/hibernatesearch"),
    @ActivationConfigProperty(propertyName="DLQMaxResent", propertyValue="1")
  } )
public class MDBSearchController extends AbstractJMSHibernateSearchController
                    implements MessageListener {
  @PersistenceContext EntityManager em;

  //method retrieving the appropriate session
  protected Session getSession() {
     return (Session) em.getDelegate();
  }

  //potentially close the session opened in #getSession(), not needed here
  protected void cleanSessionIfNeeded(Session session)
  }
}
```

이 예제는 Hibernate Search 소스 코드에서 사용할 수 있는 추상적인 Jakarta Messaging 컨트롤러 클래스를 상속하고 Jakarta EE MDB를 구현합니다. 이 구현은 예제로 제공되며 자카르타 EE 메시지 기반 빈을 사용하도록 조정할 수 있습니다.

7.2.5. Lucene 인덱싱 튜닝

7.2.5.1. Lucene 인덱싱 성능 튜닝

Hibernate Search는 **mergeFactor**, max**MergeDocs**, **maxBufferedDocs** 와 같은 기본 Lucene **IndexWriter** 로 전달되는 매개변수 세트를 지정하여 Lucene 색인 성능을 조정하는 데 사용됩니다. 이러한 매개변수를 모든 인덱스에 적용되는 기본값으로, 인덱스별로 또는 shard별로 지정합니다.

다양한 사용 사례에 맞게 조정할 수 있는 몇 가지 낮은 수준의 **IndexWriter** 설정이 있습니다. 이러한 매개변수는 **indexwriter** 키워드로 그룹화됩니다.

```
hibernate.search.[default|<indexname>].indexwriter.<parameter_name>
```

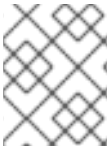
특정 shard 구성의 인덱스 라이터 값에 대한 값이 설정되지 않은 경우 Hibernate Search는 인덱스 섹션을 확인한 다음 default 섹션에서 확인합니다.

다음 테이블의 구성으로 이러한 설정이 적용되므로 두 번째 shard는 인덱스 의 두 번째 shard에 적용됩니다.

- **max_merge_docs** = 10
- **merge_factor** = 20
- **ram_buffer_size** = 64MB
- **Term_index_interval** = Lucene 기본값

다른 모든 값은 Lucene에 정의된 기본값을 사용합니다.

모든 값의 기본값은 Lucene의 자체 기본값으로 두는 것입니다. 성능 색인 및 동작 속성에 나열된 값은 사용 중인 Lucene 버전에 따라 다릅니다. 표시된 값은 버전 **2.4** 에 상대적입니다.



참고

이전 버전의 Hibernate Search에는 배치 및 트랜잭션 속성이라는 개념이 있었습니다. 백엔드가 항상 동일한 설정을 사용하여 작업을 수행하므로 이 경우에는 더 이상 그렇지 않습니다.

표 7.4. 인덱싱 성능 및 동작 속성

속성	설명	기본값
hibernate.search.[default|<indexname>].exclusive_index_use	다른 프로세스가 동일한 인덱스에 쓸 필요가 없는 경우 **true** 로 설정합니다. 이를 통해 Hibernate Search는 인덱스에서 독점 모드로 작동하고 인덱스에 변경 사항을 작성할 때 성능을 향상시킬 수 있습니다.	**true** (성능 향상, 종료 시 잠금 해제)
hibernate.search.[default|<indexname>].max_queue_length	각 인덱스에는 인덱스에 적용할 업데이트가 포함된 별도의 "파피라인"이 있습니다. 이 큐가 가득 차면 대기열에 더 많은 작업을 추가하는 작업이 차단 작업이 됩니다. **worker.execution** 을 **async** 로 구성하지 않는 한 이 설정을 구성하는 것은 바람직하지 않습니다.	**1000**
hibernate.search.[default|<indexname>].indexwriter.max_buffered_delete_terms	버퍼링된 메모리 내 삭제 조건이 적용 및 플러시되기 전에 필요한 최소한의 삭제 조건 수를 결정합니다. 당시 메모리에 버퍼링된 문서가 있으면 병합되고 새 세그먼트가 생성됩니다.	비활성화 (RAM 사용량에 따른 플러시)
hibernate.search.[default|<indexname>].indexwriter.max_buffered_docs	인덱싱 중 메모리에 버퍼링된 문서의 양을 제어합니다. RAM이 많을수록 더 커집니다.	비활성화 (RAM 사용량에 따른 플러시)
hibernate.search.[default|<indexname>].indexwriter.max_merge_docs	세그먼트에 허용되는 최대 문서 수를 정의합니다. 작은 값은 인덱스를 자주 변경하는 데 더 효과적입니다. 더 큰 값은 인덱스가 자주 변경되지 않는 경우 더 나은 검색 성능을 제공합니다.	무제한 (Integer.MAX_VALUE)
hibernate.search.[default|<indexname>].indexwriter.merge_factor	세그먼트 병합 빈도 및 크기를 제어합니다. 삽입이 발생할 때 세그먼트 인덱스가 병합되는 빈도를 결정합니다. 값이 작을수록 인덱싱하는 동안 RAM이 줄어들고 최적화되지 않은 인덱스에 대한 검색 속도가 더 빠르지만 인덱싱 속도가 느립니다. 더 큰 값을 사용하면 인덱싱 중에 더 많은 RAM이 사용되지만 최적화되지 않은 인덱스에 대한 검색 속도가 느려지지만 인덱싱이 빨라집니다. 따라서 더 큰 값(> 10)은 배치 인덱스 생성 및 대화식으로 유지 관리되는 인덱스의 경우 더 작은 값(10)에 가장 적합합니다. 값은 2보다 작아야 합니다.	10
hibernate.search.[default|<indexname>].indexwriter.merge_min_size	세그먼트 병합 빈도 및 크기를 제어합니다. 이 크기보다 작은 세그먼트(MB)는 항상 다음 세그먼트 병합 작업에 사용됩니다. 이 크기를 너무 크게 설정하면 빈번하지만 값비싼 병합 작업이 발생할 수 있습니다. **org.apache.lucene.index.LogDocMergePolicy.minMergeSize** 도 참조하십시오.	0MB (실제로 1K)

속성	설명	기본값
hibernate.search.[default|<indexname>].indexwriter.merge_max_size	세그먼트 병합 빈도 및 크기를 제어합니다. 이 크기(MB)보다 큰 세그먼트는 더 큰 세그먼트에 병합되지 않습니다. 따라서 메모리 요구 사항을 줄이고 최적의 검색 속도의 비용으로 일부 병합 작업을 방지할 수 있습니다. 인덱스를 최적화할 때 이 값은 무시됩니다. **org.apache.lucene.index.LogDocMergePolicy.maxMergeSize** 도 참조하십시오.	무제한
hibernate.search.[default|<indexname>].indexwriter.merge_max_optimize_size	세그먼트 병합 빈도 및 크기를 제어합니다. 이 크기보다 큰 세그먼트(MB)는 인덱스를 최적화하는 경우에도 더 큰 세그먼트에서 병합되지 않습니다(**merge_max_size** 설정 참조). **org.apache.lucene.index.LogDocMergePolicy.maxMergeSizeForOptimize에** 적용됩니다.	무제한
hibernate.search.[default|<indexname>].indexwriter.merge_calibrate_by_deletes	세그먼트 병합 빈도 및 크기를 제어합니다. 병합 정책을 추정할 때 삭제된 문서를 고려하지 않으려면 **false** 로 설정합니다. **org.apache.lucene.index.LogMergePolicy.calibrateSizeByDeletes** 에 적용됩니다.	**true**
hibernate.search.[default|<indexname>].indexwriter.ram_buffer_size	문서 버퍼 전용 RAM 크기(MB)를 제어합니다. max_buffered_docs를 함께 사용하면 먼저 플러시가 발생합니다. 일반적으로 더 빠른 인덱싱 성능을 위해 문서 개수 대신 RAM 사용량에 의해 플러시하고 가능한 한 큰 RAM 버퍼를 사용하는 것이 가장 좋습니다.	16MB
hibernate.search.[default|<indexname>].indexwriter.term_index_interval	색인된 용어 간 간격을 설정합니다. 큰 값은 IndexReader에서 메모리를 덜 사용하지만 용어에 대한 랜덤 액세스 속도가 느립니다.Small 값은 IndexReader에서 더 많은 메모리를 사용하고 용어에 대한 임의 액세스 속도를 높입니다. 자세한 내용은 Lucene 설명서를 참조하십시오.	128

속성	설명	기본값
hibernate.search.[default|<indexname>].indexwriter.use_compound_file	복합 파일 형식을 사용할 경우의 이점은 파일 설명자가 더 적게 사용되는 것입니다. 단점은 인덱싱에 더 많은 시간과 임시 디스크 공간이 필요하다는 것입니다. 인덱싱 시간을 개선하기 위해 이 매개변수를 **false**로 설정할 수 있지만 **mergeFactor** 도 크면 파일 설명자가 실행되지 않을 수 있습니다. 부울 매개 변수, **true** 또는 **false** 를 사용합니다. 이 옵션의 기본값은 **true** 입니다.	true
hibernate.search.enable_dirty_check	모든 엔터티 변경에 Lucene 인덱스 업데이트가 필요한 것은 아닙니다. 업데이트된 엔터티 속성(dirty 속성)이 모두 인덱싱되지 않은 경우 Hibernate Search는 재 인덱싱 프로세스를 건너뜁니다. 각 업데이트 이벤트에서 호출해야 하는 사용자 지정 **FieldBridges** 를 사용하는 경우 이 옵션을 비활성화합니다(필드 브리지가 구성된 속성이 변경되지 않았더라도). 이 최적화는 **@ClassBridge 또는 @DynamicBoost** 를 사용하는 클래스에는 적용되지 않습니다. 부울 매개 변수, **true** 또는 **false** 를 사용합니다. 이 옵션의 기본값은 **true** 입니다.	true

주의

차단 백엔드 는 프로덕션에서 사용되는 것이 아니라 인덱싱 병목 현상을 식별하는 툴로만 사용됩니다.

7.2.5.2. Lucene IndexWriter

다양한 사용 사례에 맞게 조정할 수 있는 몇 가지 낮은 수준의 IndexWriter 설정이 있습니다. 이러한 매개변수는 indexwriter 키워드로 그룹화됩니다.

```
default.<indexname>.indexwriter.<parameter_name>
```

shard 구성의 indexwriter 에 대한 값이 설정되지 않은 경우 Hibernate Search는 인덱스 섹션을 확인한 다음 default 섹션에서 확인합니다.

7.2.5.3. 성능 옵션 구성

다음 구성으로 이러한 설정이 적용되며, 두 번째 shard는 index index 의 두 번째 shard에 적용됩니다.

성능 옵션 구성 예

```
default.Animals.2.indexwriter.max_merge_docs = 10
default.Animals.2.indexwriter.merge_factor = 20
default.Animals.2.indexwriter.term_index_interval = default
default.indexwriter.max_merge_docs = 100
default.indexwriter.ram_buffer_size = 64
```

- **max_merge_docs** = 10
- **merge_factor** = 20
- **ram_buffer_size** = 64MB
- **Term_index_interval** = Lucene 기본값

다른 모든 값은 Lucene에 정의된 기본값을 사용합니다.

Lucene 기본값은 Hibernate Search의 기본 설정입니다. 따라서 다음 표에 나열된 값은 사용 중인 Lucene 버전에 따라 다릅니다. 표시된 값은 버전 **2.4** 에 상대적입니다. Lucene 색인 성능에 대한 자세한 내용은 Lucene 문서를 참조하십시오.

참고

백엔드는 항상 동일한 설정을 사용하여 작업을 수행합니다.

표 7.5. 인덱싱 성능 및 동작 속성

속성	설명	기본값
default.<indexname>.exclusive_index_use	다른 프로세스가 동일한 인덱스에 쓸 필요가 없는 경우 **true** 로 설정합니다. 이를 통해 Hibernate Search는 인덱스에서 독점 모드로 작동하고 인덱스에 변경 사항을 작성할 때 성능을 향상시킬 수 있습니다.	**true** (성능 향상, 종료 시 잠금 해제)
default.<indexname>.max_queue_length	각 인덱스에는 인덱스에 적용할 업데이트가 포함된 별도의 "파피라인"이 있습니다. 이 큐가 가득 차면 대기열에 더 많은 작업을 추가하는 작업이 차단 작업이 됩니다. **worker.execution** 을 **async** 로 구성하지 않는 한 이 설정을 구성하는 것은 바람직하지 않습니다.	**1000**
default.<indexname>.indexwriter.max_buffered_delete_terms	버퍼링된 메모리 내 삭제 조건이 적용 및 플러시되기 전에 필요한 최소한의 삭제 조건 수를 결정합니다. 당시 메모리에 버퍼링된 문서가 있으면 병합되고 새 세그먼트가 생성됩니다.	비활성화 (RAM 사용량에 따른 플러시)
default.<indexname>.indexwriter.max_buffered_docs	인덱싱 중 메모리에 버퍼링된 문서의 양을 제어합니다. RAM이 많을수록 더 커집니다.	비활성화 (RAM 사용량에 따른 플러시)

속성	설명	기본값
default.<indexname>.indexwriter.max_merge_docs	세그먼트에 허용되는 최대 문서 수를 정의합니다. 작은 값은 인덱스를 자주 변경하는 데 더 효과적입니다. 더 큰 값은 인덱스가 자주 변경되지 않는 경우 더 나은 검색 성능을 제공합니다.	무제한 (Integer.MAX_VALUE)
default.<indexname>.indexwriter.merge_factor	세그먼트 병합 빈도 및 크기를 제어합니다. 삽입이 발생할 때 세그먼트 인덱스가 병합되는 빈도를 결정합니다. 값이 작을수록 인덱싱하는 동안 RAM이 줄어들고 최적화되지 않은 인덱스에 대한 검색 속도가 더 빠르지만 인덱싱 속도가 느립니다. 더 큰 값을 사용하면 인덱싱 중에 더 많은 RAM이 사용되지만 최적화되지 않은 인덱스에 대한 검색 속도가 느려지지만 인덱싱이 빨라집니다. 따라서 더 큰 값(> 10)은 배치 인덱스 생성 및 대화식으로 유지 관리되는 인덱스의 경우 더 작은 값(10)에 가장 적합합니다. 값은 2보다 작아야 합니다.	10
default.<indexname>.indexwriter.merge_min_size	세그먼트 병합 빈도 및 크기를 제어합니다. 이 크기보다 작은 세그먼트(MB)는 항상 다음 세그먼트 병합 작업에 사용됩니다. 이 크기를 너무 크게 설정하면 빈번하지만 값비싼 병합 작업이 발생할 수 있습니다. **org.apache.lucene.index.LogDocMergePolicy.minMergeSize** 도 참조하십시오.	0MB (실제로 1K)
default.<indexname>.indexwriter.merge_max_size	세그먼트 병합 빈도 및 크기를 제어합니다. 이 크기(MB)보다 큰 세그먼트는 더 큰 세그먼트에 병합되지 않습니다. 따라서 메모리 요구 사항을 줄이고 최적의 검색 속도의 비용으로 일부 병합 작업을 방지할 수 있습니다. 인덱스를 최적화할 때 이 값은 무시됩니다. **org.apache.lucene.index.LogDocMergePolicy.maxMergeSize** 도 참조하십시오.	무제한
default.<indexname>.indexwriter.merge_max_optimize_size	세그먼트 병합 빈도 및 크기를 제어합니다. 이 크기보다 큰 세그먼트(MB)는 인덱스를 최적화하는 경우에도 더 큰 세그먼트에서 병합되지 않습니다(**merge_max_size** 설정 참조). **org.apache.lucene.index.LogDocMergePolicy.maxMergeSizeForOptimize에** 적용됩니다.	무제한

속성	설명	기본값
default.<indexname>.indexwriter.merge_calibrate_by_deletes	세그먼트 병합 빈도 및 크기를 제어합니다. 병합 정책을 추정할 때 삭제된 문서를 고려하지 않으려면 **false** 로 설정합니다. **org.apache.lucene.index.LogMergePolicy.calibrateSizeByDeletes** 에 적용됩니다.	**true**
default.<indexname>.indexwriter.ram_buffer_size	문서 버퍼 전용 RAM 크기(MB)를 제어합니다. max_buffered_docs를 함께 사용하면 먼저 플러시가 발생합니다. 일반적으로 더 빠른 인덱싱 성능을 위해 문서 개수 대신 RAM 사용량에 의해 플러시하고 가능한 한 큰 RAM 버퍼를 사용하는 것이 가장 좋습니다.	16MB
default.<indexname>.indexwriter.term_index_interval	색인된 용어 간 간격을 설정합니다. 큰 값은 IndexReader에서 메모리를 덜 사용하지만 용어에 대한 임의 액세스 속도가 느립니다. 작은 값은 IndexReader에서 더 많은 메모리를 사용하고 용어에 대한 임의 액세스 속도를 높입니다. 자세한 내용은 Lucene 설명서를 참조하십시오.	128
default.<indexname>.indexwriter.use_compound_file	복합 파일 형식을 사용할 경우의 이점은 파일 설명자가 더 적게 사용되는 것입니다. 단점은 인덱싱에 더 많은 시간과 임시 디스크 공간이 필요하다는 것입니다. 인덱싱 시간을 개선하기 위해 이 매개변수를 **false로** 설정할 수 있지만 **mergeFactor** 도 크면 파일 설명자가 실행되지 않을 수 있습니다. 부울 매개 변수, **true** 또는 **false** 를 사용합니다. 이 옵션의 기본값은 **true** 입니다.	true
default.enable_dirty_check	모든 엔터티 변경에 Lucene 인덱스 업데이트가 필요한 것은 아닙니다. 업데이트된 엔터티 속성(dirty 속성)이 모두 인덱싱되지 않은 경우 Hibernate Search는 재 인덱싱 프로세스를 건너뜁니다. 각 업데이트 이벤트에서 호출해야 하는 사용자 지정 **FieldBridges** 를 사용하는 경우 이 옵션을 비활성화합니다(필드 브리지가 구성된 속성이 변경되지 않았더라도). 이 최적화는 **@ClassBridge 또는 @DynamicBoost** 를 사용하는 클래스에는 적용되지 않습니다. 부울 매개 변수, **true** 또는 **false** 를 사용합니다. 이 옵션의 기본값은 **true** 입니다.	true

속성	설명	기본값

7.2.5.4. 인덱싱 속도 조정

아키텍처에서 이를 허용하는 경우 인덱스 쓰기 효율성을 높이기 위해 **default.exclusive_index_use=true** 를 유지합니다.

인덱싱 속도를 튜닝할 때 권장되는 접근 방식은 오브젝트 로드를 최적화하는 데 먼저 집중한 다음 달성한 타이밍을 기준으로 사용하여 인덱싱 프로세스를 조정하는 것입니다. 차단을 작업자 백엔드 로 설정하고 인덱싱 루틴을 시작합니다. 이 백엔드는 Hibernate 검색을 비활성화하지 않습니다. 필요한 변경 집합을 인덱스에 생성하지만 인덱스에 플러시하는 대신 폐기합니다. **hibernate.search.indexing_strategy** 를 **manual** 로 설정하는 것과 반대로, 연결된 엔티티도 다시 인덱싱 되므로 블랙 라인을 사용하면 데이터베이스에서 더 많은 데이터를 로드할 수 있습니다.

```
hibernate.search.[default|<indexname>].worker.backend blackhole
```

주의

차단 백엔드 는 인덱싱 병목 현상 식별을 위한 진단 도구로만 프로덕션에서 사용하지 않습니다.

7.2.5.5. 제어 세그먼트 크기

다음 옵션은 생성된 세그먼트의 최대 크기를 구성합니다.

- **merge_max_size**
- **merge_max_optimize_size**
- **merge_calibrate_by_deletes**

제어 세그먼트 크기

```
//to be fairly confident no files grow above 15MB, use:
hibernate.search.default.indexwriter.ram_buffer_size = 10
hibernate.search.default.indexwriter.merge_max_optimize_size = 7
hibernate.search.default.indexwriter.merge_max_size = 7
```

병합 세그먼트가 두 세그먼트를 하나의 큰 세그먼트로 결합하므로 병합 작업의 max_size 를 하드 제한 세그먼트 크기의 절반 미만으로 설정합니다.

새 세그먼트는 처음에 예상보다 큰 크기일 수 있지만 세그먼트는 ram_buffer_size 보다 크게 만들어지지 않습니다. 이 임계값은 추정치로 확인됩니다.

7.2.6. 잠금 구성

Lucene Directory는 Hibernate Search에서 관리하는 각 인덱스에 대해 LockingFactory 를 통해 사용자 정의 잠금 전략을 사용하여 구성할 수 있습니다.

일부 잠금 전략에는 파일 시스템 수준 잠금이 필요하며 RAM 기반 인덱스에서 사용할 수 있습니다. 이 전략을 사용할 때는 잠금 마커 파일을 저장할 파일 시스템 위치를 가리키도록 IndexBase 구성 옵션을 지정해야 합니다.

잠금 팩토리를 선택하려면 hibernate.search.<index>.locking_strategy 옵션을 다음 옵션 중 하나로 설정합니다.

- *simple*
- *네이티브*
- *단일*
- *none*

표 7.6. 사용 가능한 잠금 구현 목록

이름	클래스	설명
잠금 설정 **간단한**	org.apache.lucene.store.SimpleFSLockFactory	Java File API를 기반으로 안전하게 구현하면 마커 파일을 생성하여 인덱스 사용을 표시합니다. 어떠한 이유로 애플리케이션을 강제 종료해야 하는 경우 다시 시작하기 전에 이 파일을 제거해야 합니다.
네이티브	org.apache.lucene.store.NativeFSLockFactory	마찬가지로 **단순** 하면 마커 파일을 만들어 인덱스 사용을 표시하지만, JVM이 종료된 경우에도 잠금이 지워지도록 기본 OS 파일 잠금을 사용합니다. 이 구현에는 NFS에서 알려진 문제가 있으며 네트워크 공유에서 문제가 발생하지 않습니다. **네이티브** 는 **파일 시스템, filesystem -master 및 filesystem- slave** 디렉토리 프로바이더의 기본 구현입니다.

이름	클래스	설명
단일	org.apache.lucene.store. SingleInstanceLockFactory	이 LockFactory는 파일 마커를 사용하지 않지만 메모리에 보유된 개체 잠금입니다. 따라서 인덱스를 다른 프로세스에서 공유하지 않을 경우에만 사용할 수 있습니다. **ram** 디렉터리 프로바이더의 기본 구현입니다.
none	org.apache.lucene.store.NoLockFactory	이 인덱스의 변경 사항은 잠금에 의해 조정되지 않습니다.

다음은 잠금 전략 구성의 예입니다.

```
hibernate.search.default.locking_strategy = simple
hibernate.search.Animals.locking_strategy = native
hibernate.search.Books.locking_strategy = org.custom.components.MyLockingFactory
```

7.2.7. 인덱스 형식 호환성

Hibernate Search는 현재 이전 버전과 호환되는 API 또는 도구를 제공하여 애플리케이션을 최신 버전으로 포팅할 수 있는 기능을 제공하지 않습니다. API는 인덱스 작성 및 검색에 Apache Lucene을 사용합니다. 경우에 따라 인덱스 형식에 대한 업데이트가 필요할 수 있습니다. 이 경우 Lucene이 이전 형식을 읽을 수 없는 경우 데이터를 다시 인덱싱해야 할 가능성이 있습니다.

주의

인덱스 형식을 업데이트하기 전에 인덱스를 백업합니다.

Hibernate Search는 **hibernate.search.lucene_version** 구성 속성을 노출합니다. 이 속성은 이전 버전의 Lucene에 정의된 대로 Analyzer 및 기타 Lucene 클래스가 동작을 준수하도록 지시합니다. **lucene-core.jar**에 포함된 **org.apache.lucene. util.Version** 도 참조하십시오. 옵션을 지정하지 않으면 Hibernate Search는 Lucene에 버전 기본값을 사용하도록 지시합니다. 업그레이드가 발생할 때 자동 변경되지 않도록 사용된 버전을 구성에 명시적으로 정의하는 것이 좋습니다. 업그레이드 후 필요한 경우 구성 값을 명시적으로 업데이트할 수 있습니다.

분석기가 Lucene 3.0 생성 인덱스와 호환되도록 합니다.

```
hibernate.search.lucene_version = LUCENE_30
```

구성된 **SearchFactory** 는 전역적이며 관련 매개 변수가 포함된 모든 Lucene API에 영향을 미칩니다. Lucene을 사용하고 Hibernate Search를 우회하면 일관된 결과를 위해 동일한 값을 적용합니다.

7.3. 애플리케이션을 위한 HIBERNATE 검색

7.3.1. Hibernate 검색의 첫 번째 단계

애플리케이션을 위한 Hibernate 검색을 시작하려면 다음 주제를 따르십시오.

- Maven을 사용하여 Hibernate 검색 활성화
- 인덱싱
- 검색 중
- Analyzer

7.3.2. Maven을 사용하여 Hibernate 검색 활성화

Maven 프로젝트에서 다음 구성을 사용하여 **hibernate-search-orm** 종속성을 추가합니다.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-search-orm</artifactId>
      <version>5.5.1.Final-redhat-1</version>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-search-orm</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

7.3.3. 주석 추가

이 섹션의 경우 서적 세부 정보가 포함된 데이터베이스가 있는 예를 살펴보십시오. 애플리케이션에는 Hibernate 관리 클래스 예가 포함되어 있습니다 **.Book** 및 **example.Author** 를 통해 무료 텍스트 검색 기능을 애플리케이션에 추가하여 서적을 검색할 수 있습니다.

예제: Hibernate 검색 특정 주석을 추가하기 전에 엔터티 책 및 작성자

```
package example;
...
@Entity
public class Book {

  @Id
  @GeneratedValue
  private Integer id;

  private String title;

  private String subtitle;
```

```
  @ManyToMany
  private Set<Author> authors = new HashSet<Author>();

  private Date publicationDate;

  public Book() {}

  // standard getters/setters follow here
  ...
}
```

```
package example;
...
@Entity
public class Author {

  @Id
  @GeneratedValue
  private Integer id;

  private String name;

  public Author() {}

  // standard getters/setters follow here
  ...
}
```

이를 위해 Book 및 Author 클래스에 몇 가지 주석을 추가해야 합니다. 첫 번째 주석 **@Indexed**는 Book을 인덱싱 가능으로 표시합니다. 설계에 따르면 Hibernate Search는 지정된 엔티티의 인덱스 불활성을 보장하기 위해 인덱스에 토큰화되지 않은 ID를 저장합니다. **@DocumentId** 는 이러한 목적으로 사용할 속성을 표시하며 대부분의 경우 데이터베이스 기본 키와 동일합니다. **@DocumentId** 주석은 **@Id** 주석이 있는 경우 선택 사항입니다.

검색할 필드는 다음과 같이 표시되어야 합니다. 이 예제에서는 제목과 제목으로 시작하고 **@Field**를 사용하여 둘 다 주석을 답니다. 매개 변수 **index=Index.YES** 는 텍스트가 인덱싱되도록 하고, **analyze=Analyze.YES** 는 기본 Lucene 분석기를 사용하여 텍스트가 분석되도록 합니다. 일반적으로 분석은 문장을 개별 단어로 분할하고 잠재적으로 **'a'** 또는 'the'와 같은 공통 단어를 제외하는 것을 의미합니다. 분석기에 대해 좀 더 자세히 살펴보겠습니다. 당사가 **@Field,store=Store.NO** 내에 지정하는 세 번째 매개 변수는 실제 데이터가 인덱스에 저장되지 않도록 합니다. 이 데이터가 인덱스에 저장되거나 검색 기능과 관련이 없는지 여부. Lucene의 관점에서는 인덱스가 생성되면 데이터를 유지할 필요가 없습니다. 저장의 이점은 프로젝션을 통해 검색할 수 있다는 것입니다.

예측이 없으면 Hibernate Search는 쿼리 기준에 맞는 엔터티의 데이터베이스 식별자를 찾고 이러한 식별자를 사용하여 데이터베이스에서 관리되는 개체를 검색하기 위해 Lucene 쿼리를 실행합니다. 예측에 대한 결정은 사례에 따라 결정되어야 합니다. 기본 동작은 관리 오브젝트를 반환하는 반면, 예측은 오브젝트 배열만 반환하므로 권장됩니다. **index=Index.YES,analyze=Analyze.YES** 및 **store=Store.NO** 는 이러한 매개변수의 기본값이며 생략할 수 있습니다.

아직 논의되지 않은 또 다른 주석은 **@DateBridge** 입니다. 이 주석은 Hibernate Search의 기본 제공 필드 브리지 중 하나입니다. Lucene 인덱스는 전적으로 문자열 기반입니다. 이러한 이유로 Hibernate Search는 인덱싱된 필드의 데이터 유형을 문자열로 변환해야 하며 그 반대의 경우도 마찬가지입니다. java.util.Date를 지정된 해상도를 사용하여 문자열로 변환할 DateBridge를 포함하여 사전 정의된 브리지 범위가 제공됩니다. 자세한 내용은 Bridges 를 참조하십시오.

그러면 **@IndexedEmbeded**가 남아 있습니다. 이 주석은 소유 엔터티의 일부로 연결된 엔터

티(@ManyToMany,@*ToOne,@Embedded 및 @ElementCollection)를 색인화하는 데 사용됩니다. 이는 Lucene 인덱스 문서가 개체 관계에 대해 아무것도 모르는 플랫 데이터 구조이기 때문에 필요합니다. 작성자의 이름을 검색할 수 있도록 하려면 이름이 책 자체의 일부로 인덱싱되었는지 확인해야 합니다. @IndexedEmbedded 위에 @Indexed를 사용하여 인덱스에 포함하려는 모든 필드를 표시해야 합니다. 자세한 내용은 Embedded 및 Associated Objects를 참조하십시오.

이러한 설정은 이제 충분해야 합니다. 엔터티 매핑에 대한 자세한 내용은 Mapping an Entity를 참조하십시오.

예제: Hibernate 검색 주석 추가 후 엔터티

```
package example;
...
@Entity

public class Book {

  @Id
  @GeneratedValue
  private Integer id;

  private String title;

  private String subtitle;

  @Field(index = Index.YES, analyze=Analyze.NO, store = Store.YES)
  @DateBridge(resolution = Resolution.DAY)
  private Date publicationDate;

  @ManyToMany
  private Set<Author> authors = new HashSet<Author>();

  public Book() {
  }

  // standard getters/setters follow here
  ...
}
```

```
package example;
...
@Entity
public class Author {

  @Id
  @GeneratedValue
  private Integer id;

  private String name;

  public Author() {
  }
```

```
  // standard getters/setters follow here
  ...
}
```

7.3.4. 인덱싱

Hibernate Search는 Hibernate Core를 통해 지속, 업데이트 또는 제거된 모든 엔터티를 투명하게 색인화합니다. 그러나 데이터베이스에 이미 있는 데이터에 대한 초기 Lucene 인덱스를 생성해야 합니다. 위의 속성과 주석을 추가한 후에는 책의 초기 배치 인덱스를 트리거할 수 있습니다. 다음 코드 조각 중 하나를 사용하여 이를 달성할 수 있습니다 (도 참조).

예제: Hibernate 세션에서 인덱스 데이터 사용

```
FullTextSession fullTextSession = org.hibernate.search.Search.getFullTextSession(session);
fullTextSession.createIndexer().startAndWait();
```

예제: 자카르타 지속성을 사용하여 인덱스 데이터

```
EntityManager em = entityManagerFactory.createEntityManager();
FullTextEntityManager fullTextEntityManager =
org.hibernate.search.jpa.Search.getFullTextEntityManager(em);
fullTextEntityManager.createIndexer().startAndWait();
```

위의 코드를 실행한 후 /var/lucene/indexes/example.Book 아래에 Lucene 인덱스가 표시됩니다. Luke를 통해 이 인덱스를 검사하면 Hibernate Search가 작동하는 방식을 이해하는 데 도움이 됩니다.

7.3.5. 검색 중

검색을 실행하려면 Lucene API 또는 Hibernate Search 쿼리 DSL을 사용하여 Lucene 쿼리를 만듭니다. 쿼리를 org.hibernate.Query로 래핑하여 Hibernate API에서 필요한 기능을 가져옵니다. 다음 코드는 인덱싱된 필드에 대해 쿼리를 준비합니다. 코드를 실행하면 Books 목록이 반환됩니다.

예제: Hibernate 검색 세션을 사용하여 검색 생성 및 실행

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();

// create native Lucene query using the query DSL
// alternatively you can write the Lucene query using the Lucene query parser
// or the Lucene programmatic API. The Hibernate Search DSL is recommended though
QueryBuilder qb = fullTextSession.getSearchFactory()
    .buildQueryBuilder().forEntity( Book.class ).get();
org.apache.lucene.search.Query query = qb
  .keyword()
  .onFields("title", "subtitle", "authors.name", "publicationDate")
  .matching("Jakarta rocks!")
  .createQuery();

// wrap Lucene query in a org.hibernate.Query
org.hibernate.Query hibQuery =
    fullTextSession.createFullTextQuery(query, Book.class);

// execute search
List result = hibQuery.list();
```

```
tx.commit();
session.close();
```

예제: 자카르타 지속성을 사용하여 검색 생성 및 실행

```
EntityManager em = entityManagerFactory.createEntityManager();
FullTextEntityManager fullTextEntityManager =
   org.hibernate.search.jpa.Search.getFullTextEntityManager(em);
em.getTransaction().begin();

// create native Lucene query using the query DSL
// alternatively you can write the Lucene query using the Lucene query parser
// or the Lucene programmatic API. The Hibernate Search DSL is recommended though
QueryBuilder qb = fullTextEntityManager.getSearchFactory()
   .buildQueryBuilder().forEntity( Book.class ).get();
org.apache.lucene.search.Query query = qb
  .keyword()
  .onFields("title", "subtitle", "authors.name", "publicationDate")
  .matching("Jakarta rocks!")
  .createQuery();

// wrap Lucene query in a javax.persistence.Query
javax.persistence.Query persistenceQuery =
   fullTextEntityManager.createFullTextQuery(query, Book.class);

// execute search
List result = persistenceQuery.getResultList();

em.getTransaction().commit();
em.close();
```

7.3.6. Analyzer

인덱싱된 서적 엔티티의 제목이 리팩토링이라고 가정합니다. 기존 코드 설계를 개선하고 적중은리팩토링, 리팩토링,리팩토링 및 리팩토링 에 필요합니다. Lucene에서 인덱싱 및 검색 시 단어 줄밍을 적용하는 Analyzer 클래스를 선택합니다. Hibernate Search는 분석기를 구성하는 여러 가지 방법을 제공합니다 (자세한 내용은 Default Analyzer 및 Analyzer by Class 참조):

- 구성 파일에서 **analyzer** 속성을 설정합니다. 지정된 클래스는 기본 분석기가 됩니다.
- 엔터티 수준에서 **@Analyzer** 주석을 설정합니다.
- 필드 수준에서 **@Analyzer** 주석을 설정합니다.

정규화된 클래스 이름 또는 사용할 분석기를 지정하거나 **@Analyzer** 주석이 있는 **@AnalyzerDef** 주석에서 정의한 **Analyzerr**를 확인합니다 . 팩토리와 함께 **collectdr** 분석기 프레임워크는 후자의 옵션에 사용

됩니다. 팩토리 클래스에 대한 자세한 내용은konr JavaDoc을 참조하거나, redhat r Wiki 의 해당 섹션을 읽어보십시오.

이 예제에서는 두 개의 필터 팩토리에서 StandardTokenizerFactory를 사용합니다. LowerCaseFilterFactory 및 taintballPorterFilterFactory. 토큰자는 문장 부호 문자와 하이픈으로 단어를 분할하지만 이메일 주소와 인터넷 호스트 이름을 그대로 유지합니다. 표준 토큰화자는 이 작업 및 기타 일반 작업에 이상적입니다. 소문자로 된 필터는 토큰의 모든 문자를 소문자로 변환하고 viaball 필터는 언어별 태밍을 적용합니다.

defr 프레임워크를 사용하는 경우 임의의 수의 필터가 있는 tokenizer를 사용합니다.

예제: 분석기 정의 및 사용

```
@Indexed
@AnalyzerDef(
  name = "customanalyzer",
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
  filters = {
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = SnowballPorterFilterFactory.class,
      params = { @Parameter(name = "language", value = "English") })
 })
public class Book implements Serializable {

  @Field
  @Analyzer(definition = "customanalyzer")
  private String title;

  @Field
  @Analyzer(definition = "customanalyzer")
  private String subtitle;

  @IndexedEmbedded
  private Set authors = new HashSet();

  @Field(index = Index.YES, analyze = Analyze.NO, store = Store.YES)
  @DateBridge(resolution = Resolution.DAY)
  private Date publicationDate;

  public Book() {
  }

  // standard getters/setters follow here
  ...
}
```

@AnalyzerDef를 사용하여 분석기를 정의한 다음 @Analyzer를 사용하여 엔터티 및 속성에 적용합니다. 이 예제에서 customanalyzer 는 정의되지만 엔터티에 적용되지 않습니다. 분석기는 제목과 자막 속성에만 적용됩니다. 분석기 정의는 전역적입니다. 엔터티에 대한 분석기를 정의하고 필요에 따라 다른 엔터티에 대한 정의를 재사용합니다.

7.4. 인덱스 구조에 엔티티 매핑

7.4.1. 엔터티 매핑

인덱스 엔터티에 필요한 모든 메타데이터 정보는 주석을 통해 설명되므로 XML 매핑 파일이 필요하지 않습니다. Hibernate 매핑 파일은 기본 Hibernate 구성을 위해 계속 사용할 수 있지만, Hibernate Search의 특정 구성은 주석을 통해 표현되어야 합니다.

7.4.1.1. 기본 맵핑

엔터티 매핑에 가장 일반적으로 사용되는 주석으로 시작하겠습니다.

Lucene 기반 쿼리 API는 다음과 같은 일반적인 주석을 사용하여 엔터티를 매핑합니다.

- @Indexed
- @field
- @NumericField
- @Id

7.4.1.2. @Indexed

무엇보다도 지속 클래스를 인덱스 가능으로 선언해야 합니다. 이는 @Indexed로 클래스에 주석을 달아 수행됩니다(Indexed 로 주석 처리되지 않은 모든 엔터티는 인덱싱 프로세스에서 무시 됩니다).

```
@Entity
@Indexed
public class Essay {
...
}
```

선택적으로 **@Indexed** 주석의 **index** 속성을 지정하여 인덱스의 기본 이름을 변경할 수 있습니다.

7.4.1.3. @field

엔터티의 각 속성(또는 속성)에 대해 인덱싱 방법을 설명할 수 있습니다. 기본(주석 없음)은 인덱싱 프로세스에서 속성을 무시함을 나타냅니다.

참고

Hibernate Search 5 이전에는 숫자 필드 인코딩이 **@NumericField** 를 통해 명시적으로 요청된 경우에만 선택되었습니다. **Hibernate Search 5**부터 숫자 유형에 대해 이 인코딩이 자동으로 선택됩니다. 숫자 인코딩을 피하려면 **@Field.bridge** 또는 **@Field Bridge** 를 통해 숫자가 아닌 필드 브리지를 명시적으로 지정할 수 있습니다. **org.hibernate.search.bridge.builtin** 패키지에는 문자열을 인코딩하는 브리지 세트(예: **org.hibernate.search.bridge.builtin.IntegerBridge**)가 포함되어 있습니다.

@field 는 속성을 인덱싱으로 선언하고 다음 특성 중 하나 이상을 설정하여 인덱싱 프로세스의 여러 측면을 구성할 수 있습니다.

- **name : Lucene Document**에 저장해야 하는 이름을 설명합니다. 기본값은 속성 이름(**JavaBeans** 규칙에 따라)입니다.
- **store :** 속성이 **Lucene** 인덱스에 저장되는지 여부를 설명합니다. **Store.YES** (인덱스에 더 많은 공간이 필요하지만 예측을 허용하거나 , 압축된 방식으로 **Store.COMPRESS** (이는 더 많은 **CPU** 사용) 또는 스토리지 **Store.NO** (기본값)를 방지할 수 있습니다. 속성이 저장되면 **Lucene Document**에서 원래 값을 검색할 수 있습니다. 이는 요소가 인덱싱되었는지 여부와 관련이 없습니다.
- **index:** 속성이 인덱싱되었는지 여부를 설명합니다. 다른 값은 **Index.NO** 입니다. 즉, 색인화되지 않았으며 쿼리 및 **Index.YES**에서 찾을 수 없습니다. **YES** 는 요소가 인덱싱되어 검색 가능함을 의미합니다. 기본값은 **Index.YES** 입니다. **index.NO** 는 속성을 검색할 필요가 없지만 프로젝션에 사용할 수 있어야 하는 경우에 유용할 수 있습니다.

참고

분석 및 표준에는 인덱싱이 필요하므로 **Analyze. YES** 또는 **Norms.YES** 와 함께 **index.NO** 는 유용하지 않습니다.

- **analyze:** 속성이 분석되었는지**(Analyze.YES)**인지 여부를 결정합니다**(Analyze.NO).** 기본값은 **Analyze.YES** 입니다.

참고

속성을 분석할지 여부는 요소를 그대로 검색하는지 또는 포함된 단어로 검색할지에 따라 달라집니다. 텍스트 필드를 분석하는 것이 적합하지만 날짜 필드는 분석하지 않을 수 있습니다.

참고

정렬에 사용되는 필드는 분석할 수 *없습니다*.

- **norms:** 인덱스 시간 향상 정보를 저장해야 하는지**(Norms.YES)** 또는 사용하지 않는지**(Norms.NO)**를 설명합니다. 저장하지 않으면 상당한 양의 메모리를 절약할 수 있지만 사용 가능한 정보를 늘리는 인덱스는 없습니다. 기본값은 **Norms.YES** 입니다.

- **TermVector:** 용어 빈도 쌍의 컬렉션을 설명합니다. 이 속성을 사용하면 인덱싱 중에 문서 내에 벡터라는 용어를 저장할 수 있습니다. 기본값은 **TermVector.NO** 입니다.

이 속성의 다른 값은 다음과 같습니다.

현재의	정의
TermVector.YES	각 문서의 벡터를 저장합니다. 이렇게 하면 두 개의 동기화된 배열이 생성되고, 하나는 문서 용어를 포함하고 다른 하나는 용어의 빈도를 포함합니다.
TermVector.NO	용어 벡터를 저장하지 마십시오.
TermVector.WITH_OFFSETS	용어 벡터 및 토큰 오프셋 정보를 저장합니다. 이는 TermVector.YES와 동일하며 용어에 대한 시작 및 끝 오프셋 위치 정보가 포함됩니다.

현재의	정의
TermVector.WITH_POSITIONS	벡터 및 토큰 위치 정보 저장. TermVector.YES와 동일하며 문서의 각 용어의 서수 위치가 포함됩니다.
TermVector.WITH_POSITION_OFFSETS	용어 벡터, 토큰 위치 및 오프셋 정보를 저장합니다. 이는 YES, withTH_OFFSETS 및 withTH_POSITIONS의 조합입니다.

- **indexNullAs : 기본 null 값에 따라 인덱스가 지정되지 않습니다. 그러나 indexNullAs 를 사용하면 null 값의 토큰으로 삽입할 문자열을 지정할 수 있습니다. 기본값으로 이 값은 Field.DO_NOT_INDEX_NULL 로 설정되며 null 값이 인덱싱되지 않아야 함을 나타냅니다. 이 값을 Field.DEFAULT_NULL_TOKEN 으로 설정하여 기본 null 토큰을 사용해야 함을 나타낼 수 있습니다. 이 기본 null 토큰은 hibernate.search.default_null_token 을 사용하여 구성에서 지정할 수 있습니다. 이 속성이 설정되지 않고 Field.DEFAULT_NULL_TOKEN 문자열을 지정하면 "*null*"이 기본값으로 사용됩니다.**

참고

indexNullAs 매개 변수를 사용하는 경우 검색 쿼리에서 동일한 토큰을 사용하여 null 값을 검색하는 것이 중요합니다. 이 기능은 분석되지 않은 필드(Analyze.NO)에서만 사용하는 것이 좋습니다.

주의

사용자 정의 **FieldBridge** 또는 **TwoWayFieldBridge**를 구현할 때는 개발자가 **null** 값의 인덱싱을 처리할 수 있습니다(**LuceneOptions.indexNullAs()**의 **JavaDocs** 참조).

7.4.1.4. @NumericField

@Field 또는 @DocumentId와 동일한 범위에서 지정할 수 있는 @NumericField라는 @Field에 병행 주석이 있습니다. Integer, Long, Float 및¢ 속성에 대해 지정할 수 있습니다. 인덱스 시 트리 구조를 사용하여 값이 인덱싱됩니다. 속성이 숫자 필드로 인덱싱되면 표준 @Field 속성에서 동일한 쿼리를 수행하는 것보다 효율적인 범위 쿼리 및 정렬을 가능하게 합니다. @NumericField 주석은 다음 매개변수를 허용합니다.

현재의	정의
forField	(선택 사항) 인덱싱할 관련 @Field의 이름을 숫자로 지정합니다. 속성에 @Field 선언 이상이 포함된 경우에만 필수입니다.
precisionStep	(선택 사항) Trie 구조가 인덱스에 저장된 방식을 변경합니다. 더 작은 precisionSteps로 인해 디스크 공간 사용량이 증가하고 범위와 쿼리 정렬이 빨라집니다. 값이 클수록 사용된 공간이 줄어들고 일반 @Fields의 범위 쿼리에 더 가까운 범위 쿼리에 더 가까운 범위 쿼리가 발생합니다. 기본값은 4입니다.

@NumericField는 main, Long, Integer 및 Float만 지원합니다. 다른 숫자 유형에 대해 Lucene의 유사한 기능을 활용할 수 없으므로 나머지 유형은 기본값 또는 사용자 지정 TwoWayFieldBridge를 통해 문자열 인코딩을 사용해야 합니다.

유형 변환 중에 예상을 처리할 수 있다고 가정하여 사용자 정의 NumericFieldBridge를 사용할 수 있습니다.

예제: 사용자 지정 NumericFieldBridge 정의

```
public class BigDecimalNumericFieldBridge extends NumericFieldBridge {
  private static final BigDecimal storeFactor = BigDecimal.valueOf(100);

  @Override
  public void set(String name, Object value, Document document, LuceneOptions
luceneOptions) {
    if ( value != null ) {
      BigDecimal decimalValue = (BigDecimal) value;
      Long indexedValue = Long.valueOf( decimalValue.multiply( storeFactor ).longValue() );
      luceneOptions.addNumericFieldToDocument( name, indexedValue, document );
    }
  }

  @Override
  public Object get(String name, Document document) {
    String fromLucene = document.get( name );
    BigDecimal storedBigDecimal = new BigDecimal( fromLucene );
    return storedBigDecimal.divide( storeFactor );
  }

}
```

7.4.1.5. @Id

마지막으로 엔터티의 id (ID) 속성은 지정된 엔터티의 인덱스 고유성을 보장하기 위해 Hibernate Search에서 사용하는 특수 속성입니다. 기본적으로 ID 를 저장해야 하며 토큰화해서는 안 됩니다. 속성을 인덱스 식별자로 표시하려면 @DocumentId 주석을 사용합니다. Jakarta Persistence를 사용 중이고 @Id를 지정한 경우 @DocumentId를 생략할 수 있습니다. 선택한 엔터티 식별자도 문서 식별자로 사용됩니다.

Infinispan 쿼리에서는 엔터티의 id 속성을 사용하여 인덱스가 고유하게 식별되는지 확인합니다. 기본적으로 ID가 저장되며 토큰으로 변환해서는 안 됩니다. 속성을 인덱스 ID로 표시하려면 @DocumentId 주석을 사용합니다.

예제: 인덱스된 속성 지정

```
@Entity
@Indexed
public class Essay {
   ...
   @Id
   @DocumentId
   public Long getId() { return id; }

   @Field(name="Abstract", store=Store.YES)
   public String getSummary() { return summary; }

   @Lob
   @Field
   public String getText() { return text; }

   @Field @NumericField( precisionStep = 6)
   public float getGrade() { return grade; }
}
```

위의 예제에서는 id , Abstract,text, grade 의 네 개의 필드가 있는 인덱스를 정의합니다. 기본적으로 JavaBean 사양에 따라 필드 이름은 대문자화되지 않습니다. grade 필드에는 기본값보다 약간 큰 정확도 단계가 있는 숫자로 주석이 추가됩니다.

7.4.1.6. 매핑 속성 다중 시간

때로는 약간 다른 인덱싱 전략을 통해 인덱스당 속성을 여러 번 매핑해야 합니다. 예를 들어, 쿼리를 필드별로 정렬하려면 필드를 분석하지 않아야 합니다. 이 속성에서 단어로 검색하고 정렬하려면 인덱싱해야 합니다. 한 번 분석되지 않은 경우 분석되지 않습니다. @Fields를 사용하면 이러한 목표를 달성할 수 있습니다.

예제: @Fields를 사용하여 속성 다중 시간 매핑

```
@Entity
@Indexed(index = "Book" )
public class Book {
   @Fields( {
         @Field,
         @Field(name = "summary_forSort", analyze = Analyze.NO, store = Store.YES)
         } )
   public String getSummary() {
      return summary;
   }
   ...
}
```

이 예제에서 필드 요약 은 토큰화된 방식으로 요약 으로 한 번, 한 번은 토큰화되지 않은 방식으로 summary_forSort 로 두 번 인덱싱됩니다.

7.4.1.7. 임베디드 및 연관 개체

연결된 개체와 포함된 오브젝트는 루트 엔터티 인덱스의 일부로 인덱싱할 수 있습니다. 이는 연결된 개체의 속성을 기반으로 지정된 엔터티를 검색해야 하는 경우에 유용합니다. 목표는 연결된 도시가 애틀랜타인 장소를 반환하는 것입니다(Lucene 쿼리 구문 분석기 언어에서는 address. city:Atlanta로 변환됩니다). Place 필드가 Place 인덱스로 인덱싱됩니다. Place 인덱스 문서에는 address .id, address. street, address. treet 필드도 포함되어 있으며 쿼리할 수 있습니다.

예제: 인덱싱 연관

```
@Entity
@Indexed
public class Place {
   @Id
   @GeneratedValue
   @DocumentId
   private Long id;

   @Field
   private String name;

   @OneToOne( cascade = { CascadeType.PERSIST, CascadeType.REMOVE } )
   @IndexedEmbedded
   private Address address;
```

```
    ....
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;

    @Field
    private String street;

    @Field
    private String city;

    @ContainedIn
    @OneToMany(mappedBy="address")
    private Set<Place> places;
    ...
}
```

@IndexedEmbedded 기술을 사용할 때 **Lucene** 인덱스에서 데이터가 비정상화되므로 **Hibernate Search**는 **Place** 오브젝트의 모든 변경 사항과 인덱스를 최신 상태로 유지하기 위해 **Address** 오브젝트의 변경 사항을 알고 있어야 합니다. **Lucene** 문서가 주소 변경 시 업데이트되도록 하려면 **@ContainedIn** 과 양방향 관계의 다른 측면을 표시합니다.

참고

@ContainedIn 은 엔터티 및 포함된(수집) 개체를 가리키는 두 연결 모두에 유용합니다.

이를 확장하기 위해 다음 예제에서는 **@IndexedEmbeded** 중첩을 보여줍니다.

예제: @IndexedEmbeded 및 @ContainedIn의 중첩된 사용

```
@Entity
@Indexed
public class Place {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;
```

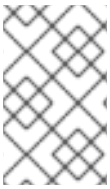
```
    @Field
    private String name;

    @OneToOne( cascade = { CascadeType.PERSIST, CascadeType.REMOVE } )
    @IndexedEmbedded
    private Address address;
    ....
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;

    @Field
    private String street;

    @Field
    private String city;

    @IndexedEmbedded(depth = 1, prefix = "ownedBy_")
    private Owner ownedBy;

    @ContainedIn
    @OneToMany(mappedBy="address")
    private Set<Place> places;
    ...
}

@Embeddable
public class Owner {
    @Field
    private String name;
   ...
}
```

@*ToMany,@*ToOne 및 @ Embeded 속성에는 @ IndexedEmbeded가 주석을 달 수 있습니다. 그러면 연결된 클래스의 속성이 기본 엔터티 인덱스에 추가됩니다. 인덱스에는 다음 필드가 포함됩니다.

- id
- name

- **address.street**
- **address.city**
- **address.ownedBy_name**

기본 접두사는 기존 오브젝트 탐색 규칙에 따라 **propertyName.** 입니다. **ownedBy** 속성에 표시되는 접두사 특성을 사용하여 재정의할 수 있습니다.



참고

접두사는 빈 문자열로 설정할 수 없습니다.

depth 속성은 오브젝트 그래프에 클래스의 재활용 종속성(인스턴스가 아님)이 포함된 경우 필요합니다. 예를 들어 소유자가 **Place**(위치)를 가리키는 경우. **Hibernate Search**는 예상 깊이에 도달한 후 인덱싱된 임베디드 속성을 포함하여 중지되거나 개체 그래프 경계에 도달합니다. 자체 참조가 있는 클래스는 재활용 종속성의 예입니다. 이 예제에서는**depth** 가 **1**로 설정되어 있으므로 소유자의 **@IndexedEmbedded** 속성이 무시됩니다.

개체 연결에 **@IndexedEmbedded** 를 사용하면 다음과 같은 쿼리(**Lucene**의 쿼리 구문 사용)를 표현할 수 있습니다.

- 이름에 **JBoss**가 포함되어 있고 주소 도시가 애틀랜타인 장소를 돌아갑니다. **Lucene** 쿼리에서는 다음과 같습니다.

```
+name:jboss +address.city:atlanta
```

- 이름에 **JBoss**가 포함되어 있고 소유자 이름에 **Joe**가 포함된 위치를 반환합니다. **Lucene** 쿼리에서는 다음과 같습니다.

```
+name:jboss +address.ownedBy_name:joe
```

이 동작은 데이터 복제 비용에 따라 보다 효율적인 방식으로 관계형 조인 작업을 모방합니다. 기본적으로 **Lucene** 인덱스에는 연관 개념이 없으며 조인 작업이 존재하지 않습니다. 전체 텍스트 인덱스 속도 및 기능 풍부성을 활용하면서 관계형 모델을 정규화하는 데 도움이 될 수 있습니다.

참고

관련된 개체는 자체적으로 (아직 할 필요는 없음) @Indexed가 될 수 있습니다.

@IndexedEmbeded 가 엔터티를 가리키는 경우 연결은 방향이어야 하며 다른 쪽에는 주석 @ContainedIn (이전 예와 같이)이 추가되어야 합니다. 그렇지 않은 경우 관련 엔터티를 업데이트할 때 Hibernate Search에서 루트 인덱스를 업데이트할 수 없습니다(이 예제에서는 연결된 Address 인스턴스를 업데이트할 때 Place 인덱스 문서를 업데이트해야 함).

@IndexedEmbeded의 주석이 추가된 개체 유형은 Hibernate 및 Hibernate Search의 대상 개체 유형이 아닌 경우도 있습니다. 이는 특히 인터페이스를 구현 대신 사용하는 경우에 해당합니다. 이러한 이유로 targetElement 매개변수를 사용하여 Hibernate Search에서 대상으로 하는 개체 유형을 재정의할 수 있습니다.

예제: targetElement Properties의 @IndexedEmbedded 사용

```
@Entity
@Indexed
public class Address {
   @Id
   @GeneratedValue
   @DocumentId
   private Long id;

   @Field
   private String street;

   @IndexedEmbedded(depth = 1, prefix = "ownedBy_", )
   @Target(Owner.class)
   private Person ownedBy;
   ...
}

@Embeddable
public class Owner implements Person { ... }
```

7.4.1.8. 특정 경로로 오브젝트 임베딩 제한

@IndexedEmbedded 주석은 깊이에 대한 대안으로 사용하거나 결합할 수 있는 includePaths 속성을 제공합니다.

임베드 유형의 모든 인덱싱된 필드만 사용하면 동일한 깊이에 반복적으로 추가됩니다. 이렇게 하면 다른 모든 필드도 추가하지 않고 특정 경로만 선택하기 어려울 수 있습니다. 이 방법은 필요하지 않을 수 있습니다.

불필요하게 로드 및 인덱싱 엔터티를 방지하려면 필요한 경로를 정확하게 지정할 수 있습니다. 일반적인 애플리케이션에는 다른 경로에 대해 서로 다른 깊이가 필요할 수 있습니다. 즉, 아래 예제와 같이 경로를 명시적으로 지정해야 할 수 있습니다.

예제: @IndexedEmbedded의 includePaths 속성 사용

```
@Entity
@Indexed
public class Person {

  @Id
  public int getId() {
    return id;
  }

  @Field
  public String getName() {
    return name;
  }

  @Field
  public String getSurname() {
    return surname;
  }

  @OneToMany
  @IndexedEmbedded(includePaths = { "name" })
  public Set<Person> getParents() {
    return parents;
  }

  @ContainedIn
  @ManyToOne
  public Human getChild() {
    return child;
  }

  ...//other fields omitted
```

위 예제와 마찬가지로 매핑을 사용하면 Person에서 이름 및/또는 성 이름 및/또는 상위 이름을 기준으

로 검색할 수 있습니다. 상위의 성 을 인덱싱하지 않으므로 부모의 성에서 검색할 수는 없지만 색인화 속도를 높이고 공간을 절약하고 전반적인 성능을 개선합니다.

@IndexedEmbeddedincludePaths에는 *일반적으로 인덱싱하는 항목 외에도* 지정된 경로가 포함됩니다. **includePaths**를 사용하고 깊이 정의되지 않은 상태로 유지되는 동작은**depth=0** 설정과 동일합니다. 포함된 경로만 인덱싱됩니다.

예제: @IndexedEmbedded의 includePaths 속성 사용

```
@Entity
@Indexed
public class Human {

  @Id
  public int getId() {
    return id;
  }

  @Field
  public String getName() {
    return name;
  }

  @Field
  public String getSurname() {
    return surname;
  }

  @OneToMany
  @IndexedEmbedded(depth = 2, includePaths = { "parents.parents.name" })
  public Set<Human> getParents() {
    return parents;
  }

  @ContainedIn
  @ManyToOne
  public Human getChild() {
    return child;
  }

   ...//other fields omitted
```

위의 예에서 모든 사람은 이름 및 성 속성을 인덱싱합니다. 심도 특성으로 인해 부모의 이름과 성도도 두 번째 행까지 인덱싱됩니다. 이름 또는 성, 직접 사람, 부모 또는 부모로 검색할 수 있습니다. 두 번째 수

준을 넘어, 우리는 색인을 하나 더 추가하지만 성이 아닌 이름만 추가할 것입니다.

이렇게 하면 인덱스에 다음 필드가 생성됩니다.

- **id:** 기본 키로
- **_hibernate_class:** 엔터티 유형 저장
- **name:** 직접 필드로
- 성**:** 직접 필드로
- **parent.name: 1** 깊이의 임베디드 필드로
- **parent.surname:** 깊이 **1**의 임베디드 필드로
- **parent.parents.name:** 깊이 **2**의 임베디드 필드로
- **parent.parents.surname:** 깊이 **2**의 임베디드 필드로
- **parent.parents.parents.name: includePaths**에 지정된 추가 경로로 사용됩니다. 첫 번째 부모. 필드 이름에서 유추되고, 나머지 경로는 **includePaths**의 속성입니다.

필요한 쿼리를 먼저 정의하여 애플리케이션을 설계하는 경우, 필요한 필드와 사용 사례를 구현하기 위해 불필요한 필드를 정확하게 알 수 있으므로 인덱싱된 경로를 명시적으로 제어할 수 있습니다.

7.4.2. boosting

Lucene은 특정 문서 또는 필드를 다른 항목보다 더 중요하거나 덜 중요할 수 있는 *향상* 이라는 개념을 가지고 있습니다. **Lucene**은 인덱스와 검색 시간 향상을 구분합니다. 다음 섹션에서는 **Hibernate Search**를 사용하여 인덱스 시간을 높일 수 있는 방법을 보여줍니다.

7.4.2.1. 정적 인덱스 시간 변경

색인화된 클래스 또는 속성에 대한 정적 **boost** 값을 정의하려면 **@Boost** 주석을 사용할 수 있습니다. **@Field** 내에서 이 주석을 사용하거나 메서드 또는 클래스 수준에서 직접 지정할 수 있습니다.

예제: @Boost 사용 방법

```
@Entity
@Indexed

public class Essay {
   ...

   @Id
   @DocumentId
   public Long getId() { return id; }

   @Field(name="Abstract", store=Store.YES, boost=@Boost(2f))
   @Boost(1.5f)
   public String getSummary() { return summary; }

   @Lob
   @Field(boost=@Boost(1.2f))
   public String getText() { return text; }

   @Field
   public String getISBN() { return isbn; }
}
```

위의 예에서 검색 목록 맨 위에 도달하는 **Essay**의 가능성은 **1.7**을 곱합니다. 요약 필드는 **isbn** 필드보다 **@Field.boost** 및 **@Boost**가 누적되므로 **3.0(2 * 1.5)**입니다. 텍스트 필드는 **isbn** 필드보다 **1.2**배 더 중요합니다. 이 설명은 가장 엄격한 용어로 잘못되어 있지만 모든 실제 목적에 대해 단순하고 현실에 근접합니다.

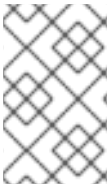
7.4.2.2. 동적 인덱스 시간 변경

Static Index Time Boosting 에 사용된 **@Boost** 주석은 런타임 시 인덱싱된 엔터티의 상태와는 별개입니다. 그러나 **boost** 요인이 엔터티의 실제 상태에 따라 달라질 수 있는 사용 사례가 있습니다. 이 경우 **@DynamicBoost** 주석을 사용자 정의 **BoostStrategy**와 함께 사용할 수 있습니다.

예제: 동적 부울

```
public enum PersonType {
   NORMAL,
   VIP
}

@Entity
@Indexed
@DynamicBoost(impl = VIPBoostStrategy.class)
public class Person {
   private PersonType type;

   // ....
}

public class VIPBoostStrategy implements BoostStrategy {
   public float defineBoost(Object value) {
      Person person = ( Person ) value;
      if ( person.getType().equals( PersonType.VIP ) ) {
         return 2.0f;
      }
      else {
         return 1.0f;
      }
   }
}
```

위 예제에서 동적 확장은 인덱스링 시 사용할 **BoostStrategy** 인터페이스의 구현으로 **VIPBoostStrategy**를 지정하는 클래스 수준에서 정의됩니다. **@DynamicBoost** 를 클래스 또는 필드 수준에서 배치할 수 있습니다. 주석 배치에 따라 전체 엔터티가 **defineBoost** 메서드 또는 주석이 지정된 필드/**property** 값에만 전달됩니다. 전달된 오브젝트를 올바른 유형으로 적용할 수 있습니다. 예에서 **VIP** 사람의 모든 인덱싱된 값은 일반 사용자의 값보다 두 배가 중요합니다.



참고

지정된 **BoostStrategy** 구현은 공용 **no-arg** 생성자를 정의해야 합니다.

물론 엔터티에서 **@Boost** 및 **@DynamicBoost** 주석을 혼합하고 일치시킬 수 있습니다. 정의된 모든 주요 요인은 누적됩니다.

7.4.3. 분석

분석은 텍스트를 단일 용어(단어)로 변환하는 프로세스이며 전체 텍스트 검색 엔진의 주요 기능 중 하나로 간주될 수 있습니다. Lucene은 분석기 개념을 사용하여 이 프로세스를 제어합니다. 다음 섹션에서는 Hibernate Search가 분석기를 구성하는 여러 가지 방법을 설명합니다.

7.4.3.1. 기본 분석기 및 분석기 클래스 별

토큰화된 필드를 색인화하는 데 사용되는 기본 analyzer 클래스는 hibernate.search.analyzer 속성을 통해 구성할 수 있습니다. 이 속성의 기본값은 org.apache.lucene.analysis.standard.StandardAnalyzer 입니다.

엔터티, 속성, 심지어 @Field별로 analyzer 클래스를 정의할 수도 있습니다(여러 필드가 단일 속성에서 인덱싱될 때 유용함).

예제: @Analyzer를 사용하는 다양한 방법

```
@Entity
@Indexed
@Analyzer(impl = EntityAnalyzer.class)
public class MyEntity {
   @Id
   @GeneratedValue
   @DocumentId
   private Integer id;

   @Field
   private String name;

   @Field
   @Analyzer(impl = PropertyAnalyzer.class)
   private String summary;

   @Field(analyzer = @Analyzer(impl = FieldAnalyzer.class)
   private String body;
   ...
}
```

이 예에서 EntityAnalyzer는 각각 PropertiesAnalyzer 및 FieldAnalyzer로 인덱싱된 요약 및 본문 을 제외하고 토큰화된 속성(이름)을 색인화하는 데 사용됩니다.

주의

동일한 엔티티에서 서로 다른 분석기를 혼합하는 것은 대부분의 경우 안 좋은 사례입니다. 쿼리 빌드를 더 복잡하게 만들고 결과를 예측할 수 없도록 합니다(초보자의 경우). 전체 쿼리에 동일한 분석기를 사용하는 경우 **QueryParser**를 사용하는 경우 더욱 복잡합니다. 지정된 필드에 대해 임박한 규칙으로 인덱싱 및 쿼리에 동일한 분석기를 사용해야 합니다.

7.4.3.2. 명명된 분석기

분석자는 처리하기에 매우 복잡해질 수 있습니다. 이러한 이유로 **Hibernate Search**를 통해 분석기 정의의 개념을 소개합니다. 분석기 정의는 많은 **@Analyzer** 선언에서 재사용할 수 있으며 다음으로 구성됩니다.

- **name:** 정의를 참조하는 데 사용되는 고유 문자열
- 문자 필터 목록: 각 **char** 필터는 토큰화 전에 입력 문자를 사전 처리합니다. 문자 필터는 문자를 추가, 변경 또는 제거할 수 있습니다. 한 가지 일반적인 사용은 문자 정규화에 사용됩니다.
- 토큰라이저: 입력 스트림을 개별 단어로 토큰화하는 역할을 합니다.
- 필터 목록: 각 필터는 토큰자가 제공하는 스트림에 단어를 제거, 수정 또는 추가하는 역할을 합니다.

이러한 작업 분리, 즉 문자 필터 목록과 필터 목록이 뒤에 오는 토큰화자는 각 개별 구성 요소를 쉽게 재사용할 수 있으며, 매우 유연한 방식으로 사용자 지정 분석기를 구축할 수 있습니다(예: **Lego**). 일반적으로 **characters** 필터는 문자 입력에서 일부 사전 처리를 수행한 다음, **Tokenizer**는 문자 입력을 토큰으로 전환한 다음 **TokenFilters**에서 추가로 처리하는 토큰화 프로세스를 시작합니다. **Hibernate Search**는 이 인프라를 지원하는데 사용됩니다. 이 프레임워크는 검색 도구 프레임워크를 활용하여 지원합니다.

아래에 명시된 구체적인 예를 검토해 보겠습니다. 먼저 이 필터는 해당 팩토리에 의해 정의됩니다. 이 예제에서는 매핑 문자 필터가 사용되며 매핑 파일에 지정된 규칙에 따라 입력에서 문자를 바꿉니다. 그런 다음 토큰라이저가 정의됩니다. 이 예에서는 표준 토큰 생성기를 사용합니다. 마지막으로, 해당 팩토리에 의해 필터 목록이 정의됩니다. 이 예제에서 **StopFilter** 필터는 전용 **words** 속성 파일을 읽습니다. 또한 필터는 대소문자를 무시해야 합니다.

예: @AnalyzerDef 및 morer Framework

```
@AnalyzerDef(name="customanalyzer",
  charFilters = {
    @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
      @Parameter(name = "mapping",
        value = "org/hibernate/search/test/analyzer/solr/mapping-chars.properties")
    })
  },
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
  filters = {
    @TokenFilterDef(factory = ISOLatin1AccentFilterFactory.class),
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = StopFilterFactory.class, params = {
      @Parameter(name="words",
        value= "org/hibernate/search/test/analyzer/solr/stoplist.properties" ),
      @Parameter(name="ignoreCase", value="true")
    })
})
public class Team {
    ...
}
```

참고

필터 및 문자 필터는 @AnalyzerDef 주석에 정의된 순서대로 적용됩니다. 주문 중요함!

일부 토큰 생성자, 토큰 필터 또는 문자 필터는 구성 또는 메타데이터 파일과 같은 리소스를 로드합니다. stop 필터와 동의어 필터의 사례입니다. resource charset이 VM 기본값을 사용하지 않는 경우 resource_charset 매개변수를 추가하여 명시적으로 지정할 수 있습니다.

예제: 특정 Charset을 사용하여 속성 파일 로드

```
@AnalyzerDef(name="customanalyzer",
  charFilters = {
    @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
      @Parameter(name = "mapping",
        value = "org/hibernate/search/test/analyzer/solr/mapping-chars.properties")
    })
  },
```

```
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
  filters = {
    @TokenFilterDef(factory = ISOLatin1AccentFilterFactory.class),
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = StopFilterFactory.class, params = {
      @Parameter(name="words",
        value= "org/hibernate/search/test/analyzer/solr/stoplist.properties" ),
      @Parameter(name="resource_charset", value = "UTF-16BE"),
      @Parameter(name="ignoreCase", value="true")
  })
})
public class Team {
    ...
}
```

정의되고 나면 다음 예제와 같이 **@Analyzer** 선언에서 분석기 정의를 재사용할 수 있습니다.

예제: 이름으로 분석기 참조

```
@Entity
@Indexed
@AnalyzerDef(name="customanalyzer", ... )
public class Team {
    @Id
    @DocumentId
    @GeneratedValue
    private Integer id;

    @Field
    private String name;

    @Field
    private String location;

    @Field
    @Analyzer(definition = "customanalyzer")
    private String description;
}
```

@AnalyzerDef에서 선언한 분석기 인스턴스는 쿼리를 작성할 때 매우 유용한 **SearchFactory**의 이름으로도 사용할 수 있습니다.

```
Analyzer analyzer = fullTextSession.getSearchFactory().getAnalyzer("customanalyzer");
```

쿼리의 필드는 공통 "언어"를 표시하도록 필드를 인덱싱하는 데 사용되는 것과 동일한 분석기를 사용하여 분석해야 합니다. 동일한 토큰은 쿼리와 인덱싱 프로세스 간에 재사용됩니다. 이 규칙에는 몇 가지 예외가 있지만 대부분의 경우 적용됩니다. 자신이 무엇을 하는지 모르는 경우라도 주의하십시오.

7.4.3.3. 사용 가능한 분석기

솔러 및 **Lucene**에는 많은 유용한 기본 문자 필터, 토큰화기 및 필터가 포함되어 있습니다. 에서 작은 필터 팩토리, 토큰 화 기 팩터 및 필터 팩토리의 전체 목록을 확인할 수 있습니다. 이 중 몇 가지를 확인하겠습니다.

표 7.7. 사용 가능한 전화 필터

팩토리	설명	매개 변수
MappingCharFilterFactory	리소스 파일에 지정된 매핑을 기반으로 하나 이상의 문자를 하나 이상의 문자로 바꿉니다.	**mapping**: "á" "a"; "ñ" ✓ "n"; "" "o"를 사용하여 매핑이 포함된 리소스 파일을 가리킵니다.
HTMLStripCharFilterFactory	HTML 표준 태그 제거, 텍스트를 유지	none

표 7.8. 사용 가능한 토큰라이저

팩토리	설명	매개 변수
StandardTokenizerFactory	Lucene StandardTokenizer 사용	none
HTMLStripCharFilterFactory	HTML 태그를 제거하고 텍스트를 유지하고 StandardTokenizer에 전달합니다.	none
PatternTokenizerFactory	지정된 정규 표현식 패턴에서 텍스트를 중단합니다.	**Pattern**: 토큰화에 사용할 정규 표현식 Group : 토큰으로 추출할 패턴 그룹을 나타냅니다.

표 7.9. 사용 가능한 필터

팩토리	설명	매개 변수
StandardFilterFactory	약어 및 단어에서 도트 제거	none
LowerCaseFilterFactory	모든 단어 소문자	none

팩토리	설명	매개 변수
StopFilterFactory	중지 단어 목록과 일치하는 단어 (토큰) 제거	Word : 중지 **단어가** 포함된 리소스 파일을 가리킵니다. **ignoreCase**: true는 중지 단어를 비교할 때 케이스를 무시해야합니다. 그렇지 않으면 false입니다.
SnowballPorterFilterFactory	지정된 언어로 단어가 루트로 줄입니다. (예: 보호, 보호, 보호는 동일한 루트를 공유합니다). 이러한 필터를 사용하면 관련 단어와 일치하는 검색이 가능합니다.	**language**: 체코어, 네덜란드어, 영어, 핀란드어, 프랑스어, 독일어, 이탈리아어, 노르웨이, 포르투갈어, 러시아어, 스페인어, 스웨덴어 등

IDE에서 org.apache.lucene.anaviv.TokenizerFactory 및 org.apache.lucene.anaoperation.TokenFilterFactory 의 모든 구현을 확인하여 사용 가능한 구현을 확인하는 것이 좋습니다.

7.4.3.4. 동적 분석기 선택

지금까지 분석기를 지정하는 모든 방법이 정적이었습니다. 그러나 색인화할 엔터티의 현재 상태에 따라 분석기를 선택하는 것이 유용한 사용 사례가 있습니다(예: 다국어 애플리케이션). 예를 들어, 블로그 Entry 클래스의 경우 분석기가 항목의 language 속성에 따라 달라질 수 있습니다. 이 속성에 따라 실제 텍스트를 색인화하도록 올바른 언어별 정형기를 선택해야 합니다.

이 동적 분석기 선택을 활성화하기 위해 Hibernate Search에는 AnalyzerDiscriminator 주석이 도입되었습니다. 다음 예제에서는 이 주석의 사용을 보여줍니다.

예제: @AnalyzerDiscriminator 사용

```
@Entity
@Indexed
@AnalyzerDefs({
  @AnalyzerDef(name = "en",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
      @TokenFilterDef(factory = LowerCaseFilterFactory.class),
      @TokenFilterDef(factory = EnglishPorterFilterFactory.class
      )
    }),
  @AnalyzerDef(name = "de",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
```

```
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = GermanStemFilterFactory.class)
  })
})
public class BlogEntry {

  @Id
  @GeneratedValue
  @DocumentId
  private Integer id;

  @Field
  @AnalyzerDiscriminator(impl = LanguageDiscriminator.class)
  private String language;

  @Field
  private String text;

  private Set<BlogEntry> references;

  // standard getter/setter
  ...
}
```

```
public class LanguageDiscriminator implements Discriminator {

  public String getAnalyzerDefinitionName(Object value, Object entity, String field) {
    if ( value == null || !( entity instanceof BlogEntry ) ) {
      return null;
    }
    return (String) value;

  }
}
```

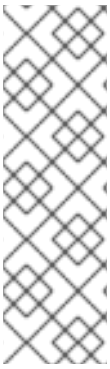
@AnalyzerDiscriminator 를 사용하기 위한 전제 조건은 동적으로 사용할 모든 분석기가 **@AnalyzerDef** 정의를 통해 사전 정의되어 있다는 것입니다. 이 경우 클래스 또는 분석기를 동적으로 선택할 엔터티의 특정 속성에 **@AnalyzerDiscriminator** 주석을 배치할 수 있습니다. **AnalyzerDiscriminator** 의 단순 매개 변수를 통해 디스크리미네이터 인터페이스의 구체적인 구현을 지정합니다. 이 인터페이스에 대한 구현을 제공해야 합니다. 구현해야 하는 유일한 메서드는 **Lucene** 문서에 추가된 각 필드에 대해 호출되는 **getAnalyzerDefinitionName()** 입니다. 인덱싱되는 엔터티도 **interface** 메서드로 전달됩니다. **value** 매개 변수는 **AnalyzerDiscriminator** 가 클래스 레벨이 아닌 속성 수준에 배치된 경우에만 설정됩니다. 이 경우 값은 이 속성의 현재 값을 나타냅니다.

기본 분석기를 재정의하지 않아야 하는 경우 디스크리미네이터 인터페이스 구현에서 기존 분석기 정의의 이름을 반환해야 합니다. 위의 예제에서는 **language** 매개변수가 **@AnalyzerDefs** 의 지정된 이름과 일치하는 **'de'** 또는 **'en'**이라고 가정합니다.

7.4.3.5. 분석기 검색

분석기 검색은 도메인 모델에서 여러 분석기를 사용했을 때 사용할 수 있습니다. 이 경우 동일한 분석기를 사용하여 쿼리를 빌드합니다. 또는 올바른 분석기를 자동으로 선택하는 **Hibernate Search** 쿼리 **DSL**을 사용합니다. 보기

Lucene 프로그래밍 방식 **API** 또는 **Lucene** 쿼리 구문 분석기를 사용하든 주어진 엔터티에 대한 범위 분석기를 검색할 수 있습니다. 범위가 지정된 분석기는 인덱싱된 필드에 따라 올바른 분석기를 적용하는 분석기입니다. 개별 필드에서 작업하는 지정된 엔터티에 여러 분석기를 정의할 수 있습니다. 범위가 지정된 분석기는 이러한 모든 분석기를 컨텍스트 인식 분석기에 통합합니다. 이론은 조금 복잡해 보이지만 쿼리에서 올바른 분석기를 사용하는 것은 매우 쉽습니다.



참고

하위 엔터티에 프로그래밍 방식의 매핑을 사용하는 경우 하위 엔터티에서 정의한 필드만 볼 수 있습니다. 상위 엔터티에서 상속된 필드 또는 메서드(@Mappedœclass로 주석이 추가됨)는 구성할 수 없습니다. 상위 엔터티에서 상속된 속성을 구성하려면 하위 엔터티에서 속성을 재정의하거나 상위 엔터티에 대한 프로그래밍 방식의 매핑을 생성합니다. 이는 하위 엔터티에 재정의되지 않는 한 상위 엔터티의 필드 또는 메서드에 주석을 달 수 없는 주석 사용을 모방합니다.

예제: 전체 텍스트 쿼리를 빌드할 때 지정된 분석기 사용

```
org.apache.lucene.queryParser.QueryParser parser = new QueryParser(
   "title",
   fullTextSession.getSearchFactory().getAnalyzer( Song.class )
);

org.apache.lucene.search.Query luceneQuery =
   parser.parse( "title:sky Or title_stemmed:diamond" );

org.hibernate.Query fullTextQuery =
   fullTextSession.createFullTextQuery( luceneQuery, Song.class );

List result = fullTextQuery.list(); //return a list of managed objects
```

위의 예에서 회색 제목은 두 가지 필드로 인덱싱됩니다. 표준 **Analyzer**는 필드 제목 에서 사용되며, 스트라이밍 분석기가 **Title_stemmed** 필드에 사용됩니다. 쿼리는 검색 팩토리에서 제공하는 **Analyzer**를 사용하여 대상 필드에 따라 적절한 분석기를 사용합니다.

참고

또한 **searchFactory.getAnalyzer(String)**를 사용하여 **@AnalyzerDef**를 통해 정의 이름으로 정의된 분석기를 검색할 수도 있습니다.

7.4.4. 브리지

엔터티에 대한 기본 매핑을 설명할 때 한 가지 중요한 팩트는 지금까지 무시되었습니다. **Lucene**에서 모든 인덱스 필드를 문자열로 표시해야 합니다. **@Field** 로 주석이 추가된 모든 엔터티 속성은 인덱싱할 문자열로 변환해야 합니다. 지금까지 언급하지 않은 이유는 **Hibernate Search**가 기본 제공 브리지 세트 덕분에 대부분의 속성에서 번역 작업을 수행하기 때문입니다. 하지만 번역 프로세스를 보다 세부적으로 제어해야 하는 경우도 있습니다.

7.4.4.1. 기본 제공 브리지

Hibernate Search는 **Java** 속성 유형과 전체 텍스트 표현 사이에 기본 제공 브리지 세트와 함께 제공됩니다.

null

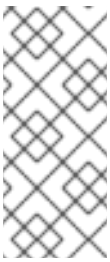
기본 **null** 요소당 인덱싱되지 않습니다. **Lucene**은 **null** 요소를 지원하지 않습니다. 그러나 경우에 따라 **null** 값을 나타내는 사용자 지정 토큰을 삽입하는 것이 유용할 수 있습니다. 자세한 내용은 를 참조하십시오.

java.lang.String

문자열은 짧음, 짧은, 정수, 정수, 긴, 긴, 온도, **Float, double,** 인덱싱됩니다.

Double, BigInteger, BigDecimal

숫자는 문자열 표현으로 변환됩니다. **Lucene**(즉, 범위가 지정된 쿼리에 사용되는) 번호는 즉시 비교할 수 없습니다. 이 번호는 반드시 채워져야 합니다.



참고

범위 쿼리에는 단점이 있습니다. 대체 방법은 결과 쿼리를 적절한 범위에 필터링하는 필터 쿼리를 사용하는 것입니다. **Hibernate Search**는 Custom Bridges 에 설명된 대로 사용자 지정 **StringBridge**도 지원합니다.

java.util.Date

날짜는 **yyyyMMddHHmmssSSS**로 저장됩니다(200611072203012(200611072203012) 2006년 11월 7일 오후 4시와 12ms EST). 내부 형식에 대해 전혀 신경 쓰지 않아야 합니다. 중요한 것은

TermRangeQuery를 사용할 때 날짜가 기본값으로 표현되어야 한다는 것입니다.

일반적으로 최대 밀리초까지 날짜를 저장할 필요는 없습니다. @DateBridge 는 인덱스 (@DateBridge(resolution=Resolution.DAY)에 저장하려는 적절한 해상도를 정의합니다. 그런 다음 날짜 패턴이 그에 따라 잘립니다.

```
@Entity
@Indexed
public class Meeting {
  @Field(analyze=Analyze.NO)

  private Date date;
  ...
```

주의

해상도가 MILLISECOND 보다 낮은 날짜는 @DocumentId 가 될 수 없습니다.

중요

기본 날짜 브리지는 Lucene의 DateTools를 사용하여 String에서 변환됩니다. 즉, 모든 날짜가 시간에 표시됨을 의미합니다. 특정 시간대에 날짜를 저장해야 하는 경우 사용자 지정 날짜 브리지를 구현해야 합니다. 최신 인덱싱 및 검색과 관련하여 귀사의 애플리케이션 요구 사항을 이해해야 합니다.

java.net.URI, java.net.URL

URI와 URL은 문자열 표현으로 변환됩니다.

java.lang.Class

클래스는 정규화된 클래스 이름으로 변환됩니다. 스레드 컨텍스트 클래스 로더는 클래스를 다시 사용할 때 사용됩니다.

7.4.4.2. 사용자 정의 브리지

경우에 따라 Hibernate Search의 기본 제공 브리지가 일부 속성 유형을 다루지 않거나 브리지에서 사용하는 문자열 표시가 요구 사항을 충족하지 않는 경우가 있습니다. 다음 단락에서는 이 문제에 대한 여러

솔루션을 설명합니다.

7.4.4.2.1. StringBridge

가장 간단한 사용자 지정 솔루션은 **Hibernate Search**에 예상되는 **Object to String** 브리지 구현을 제공하는 것입니다. 이렇게 하려면 **org.hibernate.search.bridge.StringBridge** 인터페이스를 구현해야 합니다. 모든 구현은 동시에 사용되므로 스레드 보안을 유지해야 합니다.

예제: 사용자 정의 StringBridge 구현

```
/**
 * Padding Integer bridge.
 * All numbers will be padded with 0 to match 5 digits
 *
 * @author Emmanuel Bernard
 */
public class PaddedIntegerBridge implements StringBridge {

   private int PADDING = 5;

   public String objectToString(Object object) {
      String rawInteger = ( (Integer) object ).toString();
      if (rawInteger.length() > PADDING)
         throw new IllegalArgumentException( "Try to pad on a number too big" );
      StringBuilder paddedInteger = new StringBuilder( );
      for ( int padIndex = rawInteger.length() ; padIndex < PADDING ; padIndex++ ) {
         paddedInteger.append('0');
      }
      return paddedInteger.append( rawInteger ).toString();
   }
}
```

이전 예제에서 정의된 문자열 브리지의 경우 속성 또는 필드에서 **@FieldBridge** 주석으로 이 브리지를 사용할 수 있습니다.

```
@FieldBridge(impl = PaddedIntegerBridge.class)
private Integer length;
```

7.4.4.2.2. 매개변수화된 브리지

매개 변수를 브리지 구현에 전달하여 보다 유연하게 만들 수도 있습니다. 다음 예제에서는 **ParameterizedBridge** 인터페이스를 구현하고 매개 변수는 **@FieldBridge** 주석을 통해 전달됩니다.

예제: 브리지 구현에 매개 변수 전달

```
public class PaddedIntegerBridge implements StringBridge, ParameterizedBridge {

   public static String PADDING_PROPERTY = "padding";
   private int padding = 5; //default

   public void setParameterValues(Map<String,String> parameters) {
      String padding = parameters.get( PADDING_PROPERTY );
      if (padding != null) this.padding = Integer.parseInt( padding );
   }

   public String objectToString(Object object) {
      String rawInteger = ( (Integer) object ).toString();
      if (rawInteger.length() > padding)
         throw new IllegalArgumentException( "Try to pad on a number too big" );
      StringBuilder paddedInteger = new StringBuilder( );
      for ( int padIndex = rawInteger.length() ; padIndex < padding ; padIndex++ ) {
         paddedInteger.append('0');
      }
      return paddedInteger.append( rawInteger ).toString();
   }
}


//property
@FieldBridge(impl = PaddedIntegerBridge.class,
          params = @Parameter(name="padding", value="10")
         )
private Integer length;
```

ParameterizedBridge 인터페이스는 StringBridge,TwoWayStringBridge,FieldBridge 구현을 통해 구현할 수 있습니다.

모든 구현은 스레드 보안이어야 하지만, 매개 변수는 초기화 중에 설정되며 이 단계에서 특별한 주의가 필요하지 않습니다.

7.4.4.2.3. 인식 브리지 입력

경우에 따라 브리지가 적용되는 유형을 가져오는 것이 유용할 수 있습니다.

- 필드/getter 수준 브리지의 속성 반환 유형입니다.
- 클래스 수준 브리지의 클래스 유형입니다.

예를 들어 사용자 지정 방식으로 열거를 처리하지만 실제 열거 유형에 액세스해야 하는 브리지가 있습니다. **AppliedOnTypeAwareBridge**를 구현하는 모든 브리지는 브리지가 삽입되는 유형을 가져옵니다. 매개 변수와 마찬가지로 삽입된 유형은 스레드 보안과 관련하여 특별한 주의가 필요하지 않습니다.

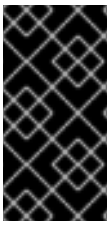
7.4.4.2.4. 두 번째 브리지

id 속성(즉, **@DocumentId** 로 주석이 추가됨)에서 브리지 구현을 사용해야 하는 경우 **2-WayString Bridge**라는 약간 확장된 버전의 **StringBridge** 를 사용해야 합니다. **Hibernate Search**는 식별자의 문자열 표현을 읽고 객체를 생성해야 합니다. **@FieldBridge** 주석을 사용하는 방식에는 차이가 없습니다.

예제: id 속성에 사용할 수 있는 TwoWayStringBridge 구현

```
public class PaddedIntegerBridge implements TwoWayStringBridge, ParameterizedBridge {

   public static String PADDING_PROPERTY = "padding";
   private int padding = 5; //default

   public void setParameterValues(Map parameters) {
      Object padding = parameters.get( PADDING_PROPERTY );
      if (padding != null) this.padding = (Integer) padding;
   }

   public String objectToString(Object object) {
      String rawInteger = ( (Integer) object ).toString();
      if (rawInteger.length() > padding)
         throw new IllegalArgumentException( "Try to pad on a number too big" );
      StringBuilder paddedInteger = new StringBuilder( );
      for ( int padIndex = rawInteger.length() ; padIndex < padding ; padIndex++ ) {
         paddedInteger.append('0');
      }
      return paddedInteger.append( rawInteger ).toString();
   }

   public Object stringToObject(String stringValue) {
      return new Integer(stringValue);
   }
}

//id property
@DocumentId
```

```
@FieldBridge(impl = PaddedIntegerBridge.class,
        params = @Parameter(name="padding", value="10")
private Integer id;
```



중요

양방향 프로세스(예: object = stringToObject(objectToString(object)))가 멱등인 것이 중요합니다.

7.4.4.2.5. FieldBridge

일부 사용 사례에서는 특성을 Lucene 인덱스에 매핑할 때 변환을 문자열에 추가하는 단순한 개체가 필요합니다. 가능한 최상의 유연성을 제공하기 위해 브리지를 FieldBridge로 구현할 수도 있습니다. 이 인터페이스는 속성 값을 제공하며 Lucene Document에서 원하는 대로 매핑할 수 있습니다. 예를 들어 속성을 두 개의 다른 문서 필드에 저장할 수 있습니다. 인터페이스는 Hibernate UserTypes 개념과 매우 유사합니다.

예제: FieldBridge 인터페이스 구현

```
/**
 * Store the date in 3 different fields - year, month, day - to ease Range Query per
 * year, month or day (eg get all the elements of December for the last 5 years).
 * @author Emmanuel Bernard
 */
public class DateSplitBridge implements FieldBridge {
   private final static TimeZone GMT = TimeZone.getTimeZone("GMT");

   public void set(String name, Object value, Document document, LuceneOptions
luceneOptions) {
      Date date = (Date) value;
      Calendar cal = GregorianCalendar.getInstance(GMT);
      cal.setTime(date);
      int year = cal.get(Calendar.YEAR);
      int month = cal.get(Calendar.MONTH) + 1;
      int day = cal.get(Calendar.DAY_OF_MONTH);

      // set year
      luceneOptions.addFieldToDocument(
         name + ".year",
         String.valueOf( year ),
         document );

      // set month and pad it if needed
      luceneOptions.addFieldToDocument(
```

```
            name + ".month",
            month < 10 ? "0" : "" + String.valueOf( month ),
            document );

        // set day and pad it if needed
        luceneOptions.addFieldToDocument(
            name + ".day",
            day < 10 ? "0" : "" + String.valueOf( day ),
            document );
    }
}

//property
@FieldBridge(impl = DateSplitBridge.class)
private Date date;
```

위의 예에서 필드는 **Document**에 직접 추가되지 않습니다. 대신 추가 기능이 **LuceneOptions** 도우미에 위임됩니다. 이 도우미는 **Store** 또는 **TermVector** 와 같이 **@Field** 에서 선택한 옵션을 적용하거나 선택한 **@Boost** 값을 적용합니다. **COMPRESS** 구현의 복잡성을 캡슐화하는 것은 특히 유용합니다. **LuceneOptions**에 위임하여 문서에 필드를 추가하는 것이 바람직하지만 아무 것도 문서 편집을 중지하지 않고 필요한 경우 **LuceneOptions**를 무시합니다.

참고

LuceneOptions와 같은 클래스는 **Lucene API**의 변경 사항으로부터 애플리케이션을 보호하고 코드를 간소화하기 위해 생성됩니다. 가능할 경우 사용하지만 더 높은 유연성이 필요한 경우 에 필요하지 않습니다.

7.4.4.2.6. ClassBridge

경우에 따라 지정된 엔터티의 두 개 이상의 속성을 결합하고 이 조합을 특정 방식으로 **Lucene** 인덱스에 인덱싱하는 것이 유용할 수 있습니다. **@ClassBridge** 및 **@ClassBridges** 주석은 속성 수준과 달리 클래스 수준에서 정의할 수 있습니다. 이 경우 사용자 지정 필드 브리지 구현은 특정 속성 대신 엔터티 인스턴스를 **value** 매개 변수로 수신합니다. 다음 예제에는 표시되지 않지만 **@ClassBridge** 는 기본 매핑 섹션에서 설명하는 **termVector** 속성을 지원합니다.

예제: 클래스 브리지 구현

```
@Entity
@Indexed
(name="branchnetwork",
        store=Store.YES,
        impl = CatFieldsClassBridge.class,
```

```
        params = @Parameter( name="sepChar", value=" " ) )
public class Department {
   private int id;
   private String network;
   private String branchHead;
   private String branch;
   private Integer maxEmployees
   ...
}

public class CatFieldsClassBridge implements FieldBridge, ParameterizedBridge {
   private String sepChar;

   public void setParameterValues(Map parameters) {
      this.sepChar = (String) parameters.get( "sepChar" );
   }

   public void set( String name, Object value, Document document, LuceneOptions
luceneOptions) {
      // In this particular class the name of the new field was passed
      // from the name field of the ClassBridge Annotation. This is not
      // a requirement. It just works that way in this instance. The
      // actual name could be supplied by hard coding it below.
      Department dep = (Department) value;
      String fieldValue1 = dep.getBranch();
      if ( fieldValue1 == null ) {
         fieldValue1 = "";
      }
      String fieldValue2 = dep.getNetwork();
      if ( fieldValue2 == null ) {
         fieldValue2 = "";
      }
      String fieldValue = fieldValue1 + sepChar + fieldValue2;
      Field field = new Field( name, fieldValue, luceneOptions.getStore(),
         luceneOptions.getIndex(), luceneOptions.getTermVector() );
      field.setBoost( luceneOptions.getBoost() );
      document.add( field );
   }
}
```

이 예에서 특정 **cat FieldsClassBridge** 가 **department** 인스턴스에 적용되고 필드 브리지에서 분기와 네트워크와 연결된 인덱스를 모두 연결합니다.

7.5. HIBERNATE SEARCH를 사용하여 LUCENE 쿼리 실행

절차

Hibernate Search는 **Lucene** 쿼리를 실행하고 **InfinispanHibernate** 세션에서 관리하는 도메인 개체를 검색할 수 있습니다. 이 검색에서는 **Hibernate** 패러다임을 벗어나지 않고 **Lucene**의 강력한 기능을 제

공하여 Hibernate의 전통적인 검색 메커니즘(HQL, 기준 쿼리, 네이티브 SQL 쿼리)에 또 다른 차원을 제공합니다.

쿼리 준비 및 실행은 다음 네 단계로 구성됩니다.

- 전체 TextSession 생성
- Hibernate QueryHibernate Search 쿼리 DSL(권장)을 사용하거나 Lucene 쿼리 API를 사용하여 Lucene 쿼리 생성
- org.hibernate.Query를 사용하여 Lucene 쿼리 래핑
- list() 또는 scroll()을 호출하여 검색 실행

쿼리 기능에 액세스하려면 FullTextSession을 사용합니다. 이 검색별 세션은 쿼리 및 인덱싱 기능을 제공하기 위해 정규 org.hibernate.Session을 래핑합니다.

예제: 전체 TextSession 생성

```
Session session = sessionFactory.openSession();
...
FullTextSession fullTextSession = Search.getFullTextSession(session);
```

Full TextSession을 사용하여 Hibernate Search 쿼리 DSL 또는 네이티브 Lucene 쿼리를 사용하여 전체 텍스트 쿼리를 구축합니다.

Hibernate Search 쿼리 DSL을 사용할 때 다음 코드를 사용하십시오.

```
final QueryBuilder b = fullTextSession.getSearchFactory().buildQueryBuilder().forEntity(
Myth.class ).get();

org.apache.lucene.search.Query luceneQuery =
```

```
    b.keyword()
      .onField("history").boostedTo(3)
      .matching("storm")
      .createQuery();

org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery );
List result = fullTextQuery.list(); //return a list of managed objects
```

대안으로 Lucene 쿼리 구문 분석기 또는 Lucene 프로그래밍 방식 API를 사용하여 Lucene 쿼리를 작성합니다.

예제: QueryParser를 사용하여 Lucene 쿼리 생성

```
SearchFactory searchFactory = fullTextSession.getSearchFactory();
org.apache.lucene.queryParser.QueryParser parser =
    new QueryParser("title", searchFactory.getAnalyzer(Myth.class) );
try {
    org.apache.lucene.search.Query luceneQuery = parser.parse( "history:storm^3" );
}
catch (ParseException e) {
    //handle parsing failure
}

org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery(luceneQuery);
List result = fullTextQuery.list(); //return a list of managed objects
```

Lucene 쿼리에 빌드된 Hibernate 쿼리는 org.hibernate.Query입니다. 이 쿼리는 HQL(Hibernate Query Language), Native 및 Criteria와 같은 기타 Hibernate 쿼리 기능과 동일한 패러다임으로 유지됩니다. list(), uniqueResult(), iterate() 및 scroll()과 같은 메서드를 쿼리와 함께 사용합니다.

Hibernate Jakarta Persistence에서 동일한 확장 기능을 사용할 수 있습니다.

예제: 자카르타 지속성을 사용하여 검색 쿼리 생성

```
EntityManager em = entityManagerFactory.createEntityManager();

FullTextEntityManager fullTextEntityManager =
    org.hibernate.search.jpa.Search.getFullTextEntityManager(em);

...
```

```
final QueryBuilder b = fullTextEntityManager.getSearchFactory()
   .buildQueryBuilder().forEntity( Myth.class ).get();

org.apache.lucene.search.Query luceneQuery =
   b.keyword()
      .onField("history").boostedTo(3)
      .matching("storm")
      .createQuery();

javax.persistence.Query fullTextQuery = fullTextEntityManager.createFullTextQuery(
luceneQuery );

List result = fullTextQuery.getResultList(); //return a list of managed objects
```

참고

이 예제에서는 **Hibernate API**가 사용되었습니다. **Full TextQuery** 를 검색하는 방식을 조정하여 **Jakarta Persistence**로 동일한 예제를 작성할 수도 있습니다.

7.5.1. 쿼리 빌드

Hibernate Search 쿼리는 **Lucene** 쿼리를 기반으로 구축되므로 사용자는 모든 **Lucene** 쿼리 유형을 사용할 수 있습니다. 쿼리가 구축되면 **Hibernate Search**는 **org.hibernate.Query**를 쿼리 조작 **API**로 사용하여 추가 쿼리 처리를 수행합니다.

7.5.1.1. Lucene API를 사용하여 Lucene 쿼리 빌드

Lucene API에서는 쿼리 구문 분석기(단순 쿼리) 또는 **Lucene** 프로그래밍 방식 **API**(복합 쿼리)를 사용합니다. **Lucene** 쿼리 빌드는 **Hibernate Search** 문서의 범위를 벗어납니다. 자세한 내용은 온라인 **Lucene** 문서 또는 ***Lucene in Action*** 또는 ***Hibernate Search in Action*** 을 참조하십시오.

7.5.1.2. Lucene 쿼리 빌드

Lucene 프로그래밍 방식 **API**를 사용하면 전체 텍스트 쿼리가 가능합니다. 그러나 **Lucene** 프로그래밍 방식 **API**를 사용하는 경우 매개 변수를 해당 문자열로 변환해야 하며 올바른 분석기도 올바른 필드에 적용해야 합니다. 예를 들어 **ngram** 분석기에서는 지정된 단어의 토큰으로 여러 **ngrams**를 사용하므로 다음과 같이 검색해야 합니다. 이 작업에 **QueryBuilder**를 사용하는 것이 좋습니다.

Hibernate Search 쿼리 **API**는 다음과 같은 주요 특성을 갖추고 있습니다.

- 메서드 이름은 영어로 되어 있습니다. 결과적으로 API 작업을 일련의 영어 구문과 지침으로 읽고 이해할 수 있습니다.

- IDE 자동 완성을 사용하면 현재 입력 접두사를 완료할 수 있으며 사용자가 올바른 옵션을 선택할 수 있습니다.

- 대체로 체인 방법 패턴을 사용합니다.

- API 작업을 쉽게 사용하고 읽을 수 있습니다.

API를 사용하려면 먼저 지정된 indexedentitytype 에 연결된 쿼리 빌더를 생성합니다. 이 QueryBuilder는 사용할 분석기 및 적용할 필드 브리지를 알고 있습니다. 여러 QueryBuilders(쿼터 루트와 관련된 각 엔터티 유형)를 생성할 수 있습니다. QueryBuilder는 SearchFactory에서 파생됩니다.

```
QueryBuilder mythQB = searchFactory.buildQueryBuilder().forEntity( Myth.class ).get();
```

지정된 필드 또는 필드에 사용된 분석기를 재정의할 수도 있습니다.

```
QueryBuilder mythQB = searchFactory.buildQueryBuilder()
   .forEntity( Myth.class )
      .overridesForField("history","stem_analyzer_definition")
   .get();
```

이제 쿼리 빌더를 사용하여 Lucene 쿼리를 빌드합니다. Lucene 프로그래밍 방식 API를 사용하여 조합한 Lucene의 쿼리 구문 분석기 또는 쿼리 개체를 Hibernate Search DSL과 함께 사용하여 생성한 사용자 정의 쿼리가 사용됩니다.

7.5.1.3. 키워드 쿼리

다음 예는 특정 단어를 검색하는 방법을 보여줍니다.

```
Query luceneQuery = mythQB.keyword().onField("history").matching("storm").createQuery();
```

표 7.10. 키워드 쿼리 매개 변수

매개변수	설명
keyword()	특정 단어를 찾으려면 이 매개 변수를 사용합니다.

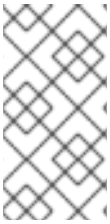
매개변수	설명
onField()	이 매개 변수를 사용하여 단어를 검색할 lucene 필드를 지정합니다.
matching()	이 매개변수를 사용하여 검색 문자열 일치 항목을 지정합니다.
createQuery()	Lucene 쿼리 개체를 생성합니다.

- "storm" 값은 기록 FieldBridge를 통해 전달됩니다. 숫자 또는 날짜가 관련된 경우 유용합니다.

- 그런 다음 필드 브리지 값이 필드 기록을 인덱싱하는 데 사용되는 분석기에 전달됩니다. 이렇게 하면 쿼리에서 인덱싱과 동일한 용어 변환(소문자, ngram, stemming 등)을 사용할 수 있습니다. 분석 프로세스에서 지정된 단어에 대한 여러 용어를 생성하는 경우 부울 쿼리가 SHOULD 논리(약 OR 논리)와 함께 사용됩니다.

유형 문자열이 아닌 속성을 검색합니다.

```
@Indexed
public class Myth {
  @Field(analyze = Analyze.NO)
  @DateBridge(resolution = Resolution.YEAR)
  public Date getCreationDate() { return creationDate; }
  public Date setCreationDate(Date creationDate) { this.creationDate = creationDate; }
  private Date creationDate;

  ...
}

Date birthdate = ...;
Query luceneQuery =
mythQb.keyword().onField("creationDate").matching(birthdate).createQuery();
```



참고

일반 Lucene에서는 Date 객체를 문자열 표현으로 변환해야 하며, 이 경우 연도입니다.

FieldBridge에 objectToString 메서드(및 모든 기본 제공 FieldBridge 구현)가 있는 경우 이 변환은 모든 오브젝트에 대해 작동합니다.

다음 예제에서는 ngram 분석기를 사용하는 필드를 검색합니다. ngram 분석기 색인의 ngram 단어의 연속으로, 사용자 오타를 방지하는 데 도움이 됩니다. 예를 들어 hibernate라는 단어의 3ms는 hib, ibe, ber, ern, rna, nat, ate입니다.

```
@AnalyzerDef(name = "ngram",
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class ),
  filters = {
    @TokenFilterDef(factory = StandardFilterFactory.class),
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = StopFilterFactory.class),
    @TokenFilterDef(factory = NGramFilterFactory.class,
      params = {
        @Parameter(name = "minGramSize", value = "3"),
        @Parameter(name = "maxGramSize", value = "3") } )
  }
)

public class Myth {
  @Field(analyzer=@Analyzer(definition="ngram")
  public String getName() { return name; }
  public String setName(String name) { this.name = name; }
  private String name;

  ...
}

Date birthdate = ...;
Query luceneQuery = mythQb.keyword().onField("name").matching("Sisiphus")
   .createQuery();
```

일치하는 단어 "Sphus"는 소문자가 지정된 후 sis, isi, sip, iph, phu, hus와 같은 3그램으로 나뉩니다. 각각의 ngram은 쿼리의 일부가 됩니다. 그런 다음 사용자는 Sysiphus myth(y 사용)를 찾을 수 있습니다. 사용자에게 투명하게 수행되는 모든 작업.

참고

사용자가 특정 필드를 필드 브리지 또는 분석기를 사용하지 않으려면 ignoreAnalyzer() 또는 ignoreFieldBridge() 함수를 호출할 수 있습니다.

동일한 필드에서 가능한 여러 단어를 검색하려면 일치하는 절에 모두 추가합니다.

```
//search document with storm or lightning in their history
Query luceneQuery =
   mythQB.keyword().onField("history").matching("storm lightning").createQuery();
```

여러 필드에서 동일한 단어를 검색하려면 **onFields** 메서드를 사용합니다.

```
Query luceneQuery = mythQB
   .keyword()
   .onFields("history","description","name")
   .matching("storm")
   .createQuery();
```

경우에 따라 동일한 용어를 검색하는 경우에도 하나의 필드를 다른 필드와 다르게 처리해야 하는 경우가 있으며, 이에 대해 **andField()** 메서드를 사용합니다.

```
Query luceneQuery = mythQB.keyword()
   .onField("history")
   .andField("name")
     .boostedTo(5)
   .andField("description")
   .matching("storm")
   .createQuery();
```

이전 예에서 필드 이름만 **5**로 늘어납니다.

7.5.1.4. 퍼지 쿼리

퍼지 쿼리(**Levenshtein** 거리 알고리즘 기반)를 실행하려면 키워드 쿼리로 시작하고 퍼지 플래그를 추가합니다.

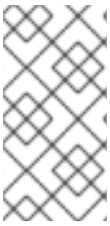
```
Query luceneQuery = mythQB
   .keyword()
     .fuzzy()
       .withThreshold( .8f )
       .withPrefixLength( 1 )
   .onField("history")
   .matching("starm")
   .createQuery();
```

임계값 은 두 개의 용어가 일치하는 것으로 간주되는 제한입니다. **0**에서 **1** 사이의 **10**진수이며 기본값은 **0.5**입니다. **prefixLength** 는 "허개"에서 무시하는 접두사 길이입니다. 기본값은 **0**이지만 다수의 고유한 용어를 포함하는 인덱스에는 **0**이 아닌 값이 권장됩니다.

7.5.1.5. 와일드카드 쿼리

와일드카드 쿼리는 단어의 일부만 알려진 경우에 유용합니다. **?** 은 단일 문자를 나타내고 *는 여러 문자를 나타냅니다. 성능상의 이유로 쿼리를 **?** 또는 * 로 시작하지 않는 것이 좋습니다.

```
Query luceneQuery = mythQB
   .keyword()
    .wildcard()
   .onField("history")
   .matching("sto*")
   .createQuery();
```



참고

와일드카드 쿼리는 일치하는 조건에 분석기를 적용하지 않습니다. * 또는 ? 의 위험은 너무 높습니다.

7.5.1.6. 구문 쿼리

지금까지 우리는 단어나 단어의 집합을 찾고 있었기 때문에 사용자는 정확하거나 대략적인 문장을 검색할 수도 있습니다. **phrase()**를 사용하여 이를 수행합니다.

```
Query luceneQuery = mythQB
   .phrase()
   .onField("history")
   .sentence("Thou shalt not kill")
   .createQuery();
```

슬롭 인수를 추가하여 대략적인 문장을 검색할 수 있습니다. 슬랩 인수는 문장에서 허용된 다른 단어 수를 나타냅니다. 이 값은 연산자 내부 또는 가까운 연산자처럼 작동합니다.

```
Query luceneQuery = mythQB
   .phrase()
    .withSlop(3)
   .onField("history")
   .sentence("Thou kill")
   .createQuery();
```

7.5.1.7. 범위 쿼리

범위 쿼리는 지정된 경계 사이의 값(포함 여부) 또는 특정 경계 이상의 값에 대한 값을 검색합니다.

```
//look for 0 <= starred < 3
Query luceneQuery = mythQB
   .range()
   .onField("starred")
   .from(0).to(3).excludeLimit()
   .createQuery();
```

```
//look for myths strictly BC
Date beforeChrist = ...;
Query luceneQuery = mythQB
    .range()
    .onField("creationDate")
    .below(beforeChrist).excludeLimit()
    .createQuery();
```

7.5.1.8. 쿼리 결합

쿼리를 결합하여 더 복잡한 쿼리를 만들 수 있습니다. 다음 집계 운영자를 사용할 수 있습니다.

- **SHOULD:** 쿼리에는 하위 쿼리의 일치하는 요소가 포함되어야 합니다.
- **MUST:** 쿼리에는 하위 쿼리의 일치하는 요소가 포함되어야 합니다.
- 쿼리에는 하위 쿼리의 일치하는 요소가 포함되어서는 안 됩니다.

하위 쿼리는 부울 쿼리 자체를 포함한 모든 **Lucene** 쿼리일 수 있습니다.

예제: SHOULD 쿼리

```
//look for popular myths that are preferably urban
Query luceneQuery = mythQB
    .bool()
      .should( mythQB.keyword().onField("description").matching("urban").createQuery() )
      .must( mythQB.range().onField("starred").above(4).createQuery() )
    .createQuery();
```

예제: 쿼리 필요

```
//look for popular urban myths
Query luceneQuery = mythQB
    .bool()
```

```
    .must( mythQB.keyword().onField("description").matching("urban").createQuery() )
    .must( mythQB.range().onField("starred").above(4).createQuery() )
  .createQuery();
```

예제: 쿼리하지 않아야 합니다

```
//look for popular modern myths that are not urban
Date twentiethCentury = ...;
Query luceneQuery = mythQB
  .bool()
    .must( mythQB.keyword().onField("description").matching("urban").createQuery() )
      .not()
    .must( mythQB.range().onField("starred").above(4).createQuery() )
    .must( mythQB
      .range()
      .onField("creationDate")
      .above(twentiethCentury)
      .createQuery() )
  .createQuery();
```

7.5.1.9. 쿼리 옵션

Hibernate Search 쿼리 DSL은 사용하기 쉽고 읽기 쉬운 쿼리 API입니다. Lucene 쿼리를 수락하고 생성할 때 DSL에서 아직 지원하지 않는 쿼리 유형을 통합할 수 있습니다.

다음은 쿼리 유형 및 필드에 대한 쿼리 옵션 요약입니다.

- boostedTo (쿼터 유형 및 현장 시)는 전체 쿼리 또는 특정 필드를 지정된 인수로 확장합니다.
- withConstantScore (쿼리 시)는 쿼리와 일치하는 모든 결과를 반환합니다.
- FilteredBy(Filter) (필터)는 Filter 인스턴스를 사용하여 쿼리 결과를 필터링합니다.

- ignoreAnalyzer (on field)는 이 필드를 처리할 때 analyzer를 무시합니다.
- ignoreFieldBridge (on field)는 이 필드를 처리할 때 필드 브리지를 무시합니다.

예제: 쿼리 옵션의 조합

```
Query luceneQuery = mythQB
    .bool()
      .should( mythQB.keyword().onField("description").matching("urban").createQuery() )
      .should( mythQB
        .keyword()
        .onField("name")
          .boostedTo(3)
          .ignoreAnalyzer()
        .matching("urban").createQuery() )
      .must( mythQB
        .range()
          .boostedTo(5).withConstantScore()
        .onField("starred").above(4).createQuery() )
    .createQuery();
```

7.5.1.10. Hibernate 검색 쿼리 빌드

7.5.1.10.1. 일반성

Lucene 쿼리를 작성한 후 Hibernate 쿼리로 래핑합니다. 쿼리는 명시적으로 구성하지 않은 경우를 제외하고 모든 인덱스된 엔터티를 검색하고 모든 유형의 색인화된 클래스를 반환합니다.

예제: Hibernate 쿼리에서 Lucene 쿼리 래핑

```
FullTextSession fullTextSession = Search.getFullTextSession( session );
org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery );
```

성능 향상을 위해 반환된 유형을 다음과 같이 제한합니다.

예제: 엔터티 유형별 검색 결과 필터링

```
fullTextQuery = fullTextSession
   .createFullTextQuery( luceneQuery, Customer.class );

// or

fullTextQuery = fullTextSession
   .createFullTextQuery( luceneQuery, Item.class, Actor.class );
```

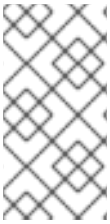
두 번째 예제의 첫 번째 부분에서는 일치하는 고객만 반환합니다. 동일한 예제의 두 번째 부분은 일치하는 행위자 및 항목을 반환합니다. 유형 제한은 다형식입니다. 그 결과 기본 클래스 **Person** 반환의 **Salesman** 및 **Customer** 두 하위 클래스가 있는 경우 결과 유형에 따라 필터링할 **Person.class**를 지정합니다.

7.5.1.10.2. 페이지 매김

성능 저하를 방지하려면 쿼리당 반환된 오브젝트 수를 제한하는 것이 좋습니다. 한 페이지에서 다른 페이지로 이동하는 사용자는 매우 일반적인 사용 사례입니다. 페이지 페이지를 정의하는 방법은 일반 **HQL** 또는 **Criteria** 쿼리에서 페이지 페이지를 정의하는 것과 유사합니다.

예제: 검색 쿼리의 페이지 번호 정의

```
org.hibernate.Query fullTextQuery =
   fullTextSession.createFullTextQuery( luceneQuery, Customer.class );
fullTextQuery.setFirstResult(15); //start from the 15th element
fullTextQuery.setMaxResults(10); //return 10 elements
```



참고

fulltextQuery.getResultSize() 를 통해 페이지 번호와 관계없이 일치하는 총 요소 수를 얻을 수 있습니다.

7.5.1.10.3. 정렬 중

Apache Lucene에는 유연하고 강력한 결과 정렬 메커니즘이 포함되어 있습니다. 기본 정렬은 관련성이 높으며 다양한 사용 사례에 적합합니다. 정렬 메커니즘은 **Lucene Sort** 개체를 사용하여 다른 속성에 따라 정렬하도록 변경하여 **Lucene** 정렬 전략을 적용할 수 있습니다.

예제: Lucene 정렬 지정

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery( query, Book.class );
org.apache.lucene.search.Sort sort = new Sort(
    new SortField("title", SortField.STRING));

List results = query.list();
```

참고

정렬에 사용되는 필드는 토큰화할 수 없습니다. 토큰화에 대한 자세한 내용은 **@Field**를 참조하십시오.

7.5.1.10.4. 전략 가져오기

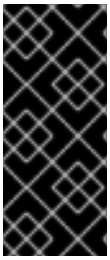
반환 유형이 하나의 클래스로 제한되면 **Hibernate Search**는 단일 쿼리를 사용하여 오브젝트를 로드합니다. **Hibernate Search**는 도메인 모델에 정의된 정적 가져오기 전략에 의해 제한됩니다. 다음과 같이 특정 사용 사례에 대한 가져오기 전략을 구체화하는 것이 유용합니다.

예제: 쿼리에 FetchMode 지정

```
Criteria criteria =
    s.createCriteria( Book.class ).setFetchMode( "authors", FetchMode.JOIN );
s.createFullTextQuery( luceneQuery ).setCriteriaQuery( criteria );
```

이 예제에서 쿼리는 **LuceneQuery**와 일치하는 모든 **Books(북)**를 반환합니다. 작성자 컬렉션은 **SQL** 외부 결합을 사용하여 동일한 쿼리에서 로드됩니다.

기준 쿼리 정의에서 제공된 기준 쿼리에 따라 유형이 추측됩니다. 따라서 반환 엔터티 유형을 제한할 필요가 없습니다.



중요

가져오기 모드는 유일한 조정 가능한 속성입니다. **getResultSize()**가 제한이 있는 **Criteria**와 함께 사용되는 경우 **SearchException**이 발생하기 때문에 **Criteria** 쿼리에 제한(위치 절)을 사용하지 마십시오.

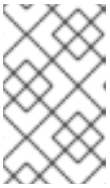
둘 이상의 엔터티가 필요한 경우 **setCrit¢Query** 를 사용하지 마십시오.

7.5.1.10.5. 프로젝션

경우에 따라 속성의 작은 하위 집합만 필요합니다. **Hibernate Search**를 사용하여 다음과 같이 속성의 하위 집합을 반환합니다.

Hibernate Search는 **Lucene** 인덱스에서 속성을 추출하고 이를 객체 표현으로 변환하고 **Object[]** 목록을 반환합니다. 예측으로 인해 시간이 소요되는 데이터베이스 왕복을 방지할 수 있습니다. 그러나 다음과 같은 제약 조건이 있습니다.

- 예상 속성은 인덱스 크기를 늘리는 인덱스(**@Field(store=Store.YES)**)에 저장해야 합니다.
- 예상 속성은 **org.hibernate.search.bridge.TwoWayFieldBridge** 또는 **org.hibernate.search.bridge.TwoWayStringBridge**를 구현하는 **Field Bridge** 를 사용해야 하며 후자는 더 간단한 버전입니다.

  참고

  모든 **Hibernate Search** 내장 유형은 양방향입니다.

- 인덱싱된 엔터티 또는 포함된 연결의 간단한 속성만 예상할 수 있습니다. 따라서 포함된 엔터티 전체를 예상할 수 없습니다.
- **@IndexedEmbedded**를 통해 인덱싱된 컬렉션 또는 맵에서는 예측이 작동하지 않습니다.

Lucene은 쿼리 결과에 대한 메타데이터 정보를 제공합니다. 예측 상수를 사용하여 메타데이터 검색.

예제: Projection을 사용하여 메타 데이터 검색

```
org.hibernate.search.FullTextQuery query =
   s.createFullTextQuery( luceneQuery, Book.class );
query.;
List results = query.list();
Object[] firstResult = (Object[]) results.get(0);
float score = firstResult[0];
Book book = firstResult[1];
String authorName = firstResult[2];
```

필드는 다음과 같은 예측 상수와 혼합할 수 있습니다.

- Full TextQuery.THIS: 초기화 및 관리 엔터티를 반환합니다(예측이 아닌 쿼리를 수행했을 때).
- FullTextQuery.DOCUMENT: 예상된 오브젝트와 관련된 Lucene 문서를 반환합니다.
- FullTextQuery.OBJECT_CLASS: 인덱싱된 엔터티의 클래스를 반환합니다.
- FullTextQuery.SCORE: 쿼리에 문서 점수를 반환합니다. 점수는 지정된 쿼리의 결과를 다른 쿼리와 비교하는 것이 유용하지만 다른 쿼리 결과를 비교할 때는 사용되지 않습니다.
- FullTextQuery.ID: 예상 오브젝트의 ID 속성 값입니다.
- FulltextQuery.DOCUMENT_ID: Lucene 문서 ID. 이 값을 Lucene 문서 ID로 사용하면 두 다른 IndexReader 열기 사이에 시간이 지남에 따라 변경될 수 있습니다.
- FullTextQuery.EXPLANATION: 지정된 쿼리에서 일치하는 개체/문서에 대한 Lucene 설명 개체를 반환합니다. 이는 대량의 데이터를 검색하는 데 적합하지 않습니다. 일반적으로 일치하는

요소당 **Lucene** 쿼리를 실행하는 데 드는 비용이 많이 듭니다. 그 결과 예측이 권장됩니다.

7.5.1.10.6. 오브젝트 초기화 전략 사용자 정의

기본적으로 **Hibernate Search**는 가장 적절한 전략을 사용하여 전체 텍스트 쿼리와 일치하는 엔터티를 초기화합니다. 필요한 엔터티를 검색하기 위해 하나 이상의 쿼리를 실행합니다. 이 접근법은 검색된 엔터티가 지속성 컨텍스트(세션) 또는 두 번째 수준 캐시에 있는 데이터베이스 이동을 최소화합니다.

두 번째 수준 캐시에 엔터티가 있는 경우 데이터베이스 개체를 검색하기 전에 **Hibernate Search**가 캐시를 검색하도록 강제 적용합니다.

예제: 쿼리를 사용하기 전에 두 번째 수준 캐시 확인

```
FullTextQuery query = session.createFullTextQuery(luceneQuery, User.class);
query.initializeObjectWith(
   ObjectLookupMethod.SECOND_LEVEL_CACHE,
   DatabaseRetrievalMethod.QUERY
);
```

ObjectLookupMethod 는(데이터베이스에서 가져오지 않고) 오브젝트에 쉽게 액세스할 수 있는지 확인하는 전략을 정의합니다. 다른 옵션은 다음과 같습니다.

- 일치하는 많은 엔터티가 이미 지속성 컨텍스트(세션 또는 **EntityManager**에 로드됨)에 로드된 경우 **ObjectLookupMethod.PERSISTENCE_CONTEXT** 가 사용됩니다.
- **ObjectLookupMethod.SECOND_LEVEL_CACHE** 는 지속성 컨텍스트 및 두 번째 수준 캐시를 확인합니다.

두 번째 수준 캐시에서 검색하도록 다음을 설정합니다.

- 두 번째 수준 캐시를 올바르게 구성하고 활성화합니다.
- 관련 엔터티에 대해 두 번째 수준 캐시를 활성화합니다. 이는 **@Cacheable**과 같은 주석을

사용하여 수행됩니다.

- Session, EntityManager 또는 Query에 대해 두 번째 수준 캐시 읽기 액세스를 활성화합니다. Hibernate 네이티브 API 또는 자카르타 지속성에서 CacheMode.NORMAL 을 사용합니다.

주의

두 번째 수준 캐시 구현이 Infinispan인 경우가 아니면 ObjectLookupMethod.SECOND_LEVEL_CACHE를 사용하지 마십시오. 다른 두 번째 수준 캐시 프로바이더는 이 작업을 효율적으로 구현하지 않습니다.

다음과 같이 DatabaseRetmastervalMethod를 사용하여 데이터베이스에서 오브젝트를 로드하는 방법을 사용자 지정합니다.

- QUERY (기본값)는 쿼리 집합을 사용하여 각 배치의 여러 오브젝트를 로드합니다. 이 방법은 권장됩니다.

- FIND_BY_ID 는 Session.get 또는 EntityManager. find 의미 체계를 사용하여 한 번에 하나의 오브젝트를 로드합니다. 이는 Hibernate Core가 배치로 엔터티를 로드할 수 있도록 배치 크기가 엔터티에 대해 설정된 경우 권장됩니다.

7.5.1.10.7. 쿼리 시간 제한

다음과 같이 Hibernate Guide에서 쿼리에 걸리는 시간을 제한합니다.

- 제한에 도착하면 예외를 발생시킵니다.

- 시간 제한이 발생할 때 검색된 결과 수로 제한합니다.

7.5.1.10.8. 시간 제한에서 예외 발생

쿼리가 정의된 시간보다 많은 시간을 사용하는 경우 QueryTimeoutException이 발생합니다 (org.hibernate.QueryTimeoutException 또는 javax.persistence.QueryTimeoutException).

기본 **Hibernate API**를 사용할 때 제한을 정의하려면 다음 방법 중 하나를 사용합니다.

예제: 쿼리 실행에서 시간 제한 정의

```
Query luceneQuery = ...;
FullTextQuery query = fullTextSession.createFullTextQuery(luceneQuery, User.class);

//define the timeout in seconds
query.setTimeout(5);

//alternatively, define the timeout in any given time unit
query.setTimeout(450, TimeUnit.MILLISECONDS);

try {
   query.list();
}
catch (org.hibernate.QueryTimeoutException e) {
   //do something, too slow
}
```

getResultSize(), **iterate()** 및 **scroll()**은 메서드 호출이 끝날 때까지 시간 제한을 준수합니다. 결과적으로 **Iterable** 또는 **ScrollableResults**는 시간 초과를 무시합니다. 또한 **explain()**은 이 시간 초과 기간을 준수하지 않습니다. 이 메서드는 디버깅에 사용되며 쿼리 성능 저하를 확인하는 데 사용됩니다.

다음은 **Jakarta Persistence**를 사용하여 실행 시간을 제한하는 표준 방법입니다.

예제: 쿼리 실행에서 시간 제한 정의

```
Query luceneQuery = ...;
FullTextQuery query = fullTextEM.createFullTextQuery(luceneQuery, User.class);

//define the timeout in milliseconds
query.setHint( "javax.persistence.query.timeout", 450 );

try {
   query.getResultList();
}
```

```
catch (javax.persistence.QueryTimeoutException e) {
   //do something, too slow
}
```

중요

예제 코드는 쿼리가 지정된 결과 양에서 중지된다는 것을 보장하지 않습니다.

7.5.2. 결과 검색

Hibernate 쿼리를 빌드한 후 HQL 또는 Criteria 쿼리와 동일한 방식으로 실행됩니다. Lucene 쿼리 쿼리에 동일한 패러다임 및 객체 의미는 적용되며 list(), uniqueResult(), 반복() 및 scroll() 과 같은 일반적인 작업을 사용할 수 있습니다.

7.5.2.1. 성능 고려 사항

적절한 수의 결과(예: 페이지 번호 사용)가 예상하고 모두 작업할 것으로 예상되는 경우 list() 또는 uniqueResult() 를 사용하는 것이 좋습니다. list() 는 엔터티 batch-size 가 올바르게 설정된 경우 가장 효과적입니다. Hibernate Search는 list(), uniqueResult() 및 반복() 을 사용할 때 Lucene Hits 요소(p Page 내에)를 처리해야 합니다.

Lucene 문서 로드를 최소화하려면 scroll() 가 더 적합합니다. 완료 시 Lucene 리소스를 유지하므로 ScrollableResults 오브젝트를 종료해야 합니다. 스크롤을 사용할 것으로 예상되지만 배치로 오브젝트를 로드하려는 경우 query.setFetchSize() 를 사용할 수 있습니다. 개체에 액세스하고 이미 로드되지 않은 경우 Hibernate Search는 다음 fetchSize 개체를 한 번에 로드합니다.

중요

스크롤보다 페이지 매김이 선호됩니다.

7.5.2.2. 결과 크기

일치하는 문서의 총 수를 아는 것이 유용할 수 있습니다.

- Google 검색에서 제공하는 대로 전체 검색 결과 기능을 제공합니다. 예를 들어 "약 888,000,000건의 결과 중 1-10회"

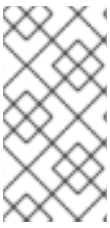
- 빠른 페이지 페이지 탐색을 구현하려면

- 제한된 쿼리가 0을 반환하거나 충분하지 않은 경우 예상을 추가하는 다단계 검색 엔진을 구현하기 위해

물론 일치하는 모든 문서를 검색하는 데 비용이 너무 많이 듭니다. Hibernate Search를 사용하면 페이지 번호 매개 변수에 관계없이 일치하는 문서 수를 검색할 수 있습니다. 더욱 흥미로운 점은 단일 오브젝트 로드를 트리거하지 않고도 일치하는 요소 수를 검색할 수 있습니다.

예제: 쿼리의 결과 크기 확인

```
org.hibernate.search.FullTextQuery query =
   s.createFullTextQuery( luceneQuery, Book.class );
//return the number of matching books without loading a single one
assert 3245 == ;

org.hibernate.search.FullTextQuery query =
   s.createFullTextQuery( luceneQuery, Book.class );
query.setMaxResult(10);
List results = query.list();
//return the total number of matching books regardless of pagination
assert 3245 == ;
```



참고

Google과 마찬가지로 인덱스가 데이터베이스(예: 비동기 클러스터)에서 완전히 최신 상태가 아닐 경우 결과 수가 예상됩니다.

7.5.2.3. ResultTransformer

예측 결과가 오브젝트 배열로 반환됩니다. 오브젝트에 사용된 데이터 구조가 애플리케이션의 요구 사항과 일치하지 않는 경우 ResultTransformer를 적용합니다. ResultTransformer는 쿼리 실행 후 필요한 데이터 구조를 빌드합니다.

예제: 예상과 함께 ResultTransformer 사용

-

```
org.hibernate.search.FullTextQuery query =
   s.createFullTextQuery( luceneQuery, Book.class );
query.setProjection( "title", "mainAuthor.name" );

query.setResultTransformer( new StaticAliasToBeanResultTransformer( BookView.class,
"title", "author" ) );
List<BookView> results = (List<BookView>) query.list();
for(BookView view : results) {
   log.info( "Book: " + view.getTitle() + ", " + view.getAuthor() );
}
```

ResultTransformer 구현의 예는 **Hibernate Core** 코드베이스에서 확인할 수 있습니다.

7.5.2.4. 결과 이해

쿼리 결과가 예상과 일치하지 않으면 **Luke** 툴에서 결과를 이해하는 데 유용합니다. 그러나 **Hibernate Search**를 사용하면 주어진 쿼리에서 **Lucene Explanation** 개체에 액세스할 수도 있습니다. 이 클래스는 **Lucene** 사용자에게 상당히 발전하지만 개체의 점수에 대해 잘 이해할 수 있습니다. 주어진 결과에 대해 설명 개체에 액세스하는 두 가지 방법이 있습니다.

- **fullTextQuery.explain(int)** 메서드 사용
- 예측 사용

첫 번째 접근 방식은 문서 **ID**를 매개 변수로 사용하고 **Explanation** 오브젝트를 반환합니다. 문서 **ID**는 예측 및 **FullTextQuery.DOCUMENT_ID** 상수를 사용하여 검색할 수 있습니다.

주의

Document ID는 엔터티 **ID**와 관련이 없습니다. 이러한 개념을 혼동하지 않도록 주의하십시오.

두 번째 접근 방식에서는 **Full TextQuery.EXPLANATION** 상수를 사용하여 **Explanation** 오브젝트를 예상합니다.

예제: Projection을 사용하여 Lucene 설명 개체 검색

```
FullTextQuery ftQuery = s.createFullTextQuery( luceneQuery, Dvd.class )
        .setProjection(
            FullTextQuery.DOCUMENT_ID,
            ,
            FullTextQuery.THIS );
@SuppressWarnings("unchecked") List<Object[]> results = ftQuery.list();
for (Object[] result : results) {
    Explanation e = (Explanation) result[1];
    display( e.toString() );
}
```

Explanation 오브젝트는 Lucene 쿼리를 다시 실행하는 것처럼 필요한 경우에만 사용합니다.

7.5.2.5. 필터

Apache Lucene에는 사용자 지정 필터링 프로세스에 따라 쿼리 결과를 필터링할 수 있는 강력한 기능이 있습니다. 이 방법은 특히 필터를 캐시하고 재사용할 수 있으므로 추가 데이터 제한 사항을 적용하는 매우 강력한 방법입니다. 사용 사례는 다음과 같습니다.

- 보안
- 시간 데이터 (예: 지난 달의 데이터 만보기)
- 컨피규레이션 필터 (예: 특정 카테고리로 검색)

Hibernate Search는 투명하게 캐시된 매개변수 지정 필터의 개념을 도입하여 개념을 추가로 푸시합니다. Hibernate Core 필터라는 개념을 잘 알고 있는 사용자의 경우 API는 매우 유사합니다.

예제: 쿼리에 대한 전체 텍스트 필터 활성화

```
fullTextQuery = s.createFullTextQuery( query, Driver.class );
fullTextQuery.enableFullTextFilter("bestDriver");
fullTextQuery.enableFullTextFilter("security").setParameter( "login", "andre" );
fullTextQuery.list(); //returns only best drivers where andre has credentials
```

이 예제에서는 쿼리 상단에 두 개의 필터를 활성화했습니다. 원하는 만큼 필터를 활성화하거나 비활성화할 수 있습니다.

필터 선언은 @FullTextFilterDef 주석을 통해 수행됩니다. 이 주석은 나중에 필터가 적용되는 쿼리와 관계없이 모든 @Indexed 엔터티에 있을 수 있습니다. 이는 필터 정의가 전역적이며 해당 이름은 고유해야 함을 의미합니다. 이름이 같은 두 개의 다른 @FullTextFilterDef 주석이 정의된 경우 SearchException이 발생합니다. 명명된 각 필터는 실제 필터 구현을 지정해야 합니다.

예제: 필터 정의 및 구현

```
@FullTextFilterDefs( {
   @FullTextFilterDef(name = "bestDriver", impl = BestDriversFilter.class),
   @FullTextFilterDef(name = "security", impl = SecurityFilterFactory.class)
})
public class Driver { ... }
```

```
public class BestDriversFilter extends org.apache.lucene.search.Filter {

   public DocIdSet getDocIdSet(IndexReader reader) throws IOException {
      OpenBitSet bitSet = new OpenBitSet( reader.maxDoc() );
      TermDocs termDocs = reader.termDocs( new Term( "score", "5" ) );
      while ( termDocs.next() ) {
         bitSet.set( termDocs.doc() );
      }
      return bitSet;
   }
}
```

BestDriversFilter는 점수가 5인 드라이버로 설정된 결과를 줄이는 간단한 Lucene 필터의 예입니다. 이 예제에서 지정된 필터는 org.apache.lucene.search.Filter 를 직접 구현하고 no-arg 생성자를 포함합니다.

필터 생성에 추가 단계가 필요한 경우 또는 사용하려는 필터에 **no-arg** 생성자가 없는 경우 팩토리 패턴을 사용할 수 있습니다.

예제: 팩토리 패턴을 사용하여 필터 생성

```
@FullTextFilterDef(name = "bestDriver", impl = BestDriversFilterFactory.class)
public class Driver { ... }

public class BestDriversFilterFactory {

@Factory
   public Filter getFilter() {
      //some additional steps to cache the filter results per IndexReader
      Filter bestDriversFilter = new BestDriversFilter();
      return new CachingWrapperFilter(bestDriversFilter);
   }
}
```

Hibernate Search는 **@Factory** 주석이 있는 메서드를 찾아 필터 인스턴스를 빌드하는 데 사용합니다. 이 팩토리에는 무중단 생성자가 있어야 합니다.

Infinispan 쿼리에서는 **@Factory** 주석이 추가된 메서드를 사용하여 필터 인스턴스를 빌드합니다. 이 팩토리에는 인수 생성자가 없어야 합니다.

명명된 필터는 매개 변수를 필터로 전달해야 하는 경우에 유용합니다. 예를 들어 보안 필터는 적용하려는 보안 수준을 알고자 할 수 있습니다.

예제: 정의된 필터에 매개 변수 전달

```
fullTextQuery = s.createFullTextQuery( query, Driver.class );
fullTextQuery.enableFullTextFilter("security").setParameter( "level", 5 );
```

각 매개변수 이름에는 대상 명명된 필터 정의의 필터 또는 필터 팩터에 연결된 **setter**가 있어야 합니다.

예제: 실제 필터 구현에서 매개 변수 사용

```
public class SecurityFilterFactory {
   private Integer level;

   /**
    * injected parameter
    */
   public void setLevel(Integer level) {
      this.level = level;
   }

   @Key public FilterKey getKey() {
      StandardFilterKey key = new StandardFilterKey();
      key.addParameter( level );
      return key;
   }

   @Factory
   public Filter getFilter() {
      Query query = new TermQuery( new Term("level", level.toString() ) );
      return new CachingWrapperFilter( new QueryWrapperFilter(query) );
   }
}
```

주석이 추가된 **@Key** 메서드는 **FilterKey** 오브젝트를 반환합니다. 반환된 오브젝트에는 특별한 계약이 있습니다. **key** 오브젝트는 **equals() / hashCode()**를 구현해야 하므로, 지정된 **Filter** 유형이 동일하고 매개 변수 집합이 동일한 경우에만 두 키가 동일할 수 있습니다. 즉, 두 개의 필터 키는 생성되는 필터를 교환할 수 있는 경우에만 두 개의 필터 키가 동일하며. 키 오브젝트는 캐시 메커니즘의 키로 사용됩니다.

@key 메서드는 다음과 같은 경우에만 필요합니다.

- 필터 캐싱 시스템이 활성화되어 있습니다 (기본적으로 활성화됨)
- 필터에 매개 변수가 있습니다.

대부분의 경우 **StandardFilterKey** 구현을 충분히 사용할 수 있습니다. 이는 **equals() / hashcode()** 구현을 각 매개 변수와 동일한 값과 해시 코드 메서드에 위임합니다.

정의된 필터가 기본 캐시당이고 캐시는 하드 및 소프트 참조의 조합을 사용하여 필요한 경우 메모리를 폐기할 수 있습니다. 하드 참조 캐시는 가장 최근에 사용된 필터를 추적하고 필요한 경우 가장 적게 사용된 필터를 변환합니다. 하드 참조 캐시의 제한에 도달하면 추가 필터가 **softReferences**로 캐시됩니다. 하드 참조 캐시의 크기를 조정하려면 **hibernate.search.filter.cache_strategy.size (기본값: 128)**를 사용합니다. 필터 캐싱의 고급 사용을 위해 자체 **FilterCachingStrategy**를 구현합니다. 클래스 이름은 **hibernate.search.filter.cache_strategy** 에서 정의합니다.

이 필터 캐싱 메커니즘은 실제 필터 결과를 캐싱하는 것과 혼동해서는 안 됩니다. **Lucene**에서는 **CachingWrapperFilter**를 중심으로 **IndexReader**를 사용하여 필터를 래핑하는 것이 일반적입니다. 래퍼는 값비싼 **recomputation**을 방지하기 위해 **getDocIdSet(IndexReader reader)** 방법에서 반환된 **DocIdSet**을 캐시합니다. 계산된 **DocIdSet**은 동일한 **IndexReader** 인스턴스에만 사용할 수 있다는 사실을 언급하는 것이 중요합니다. 리더는 열려 있는 현재 인덱스의 상태를 효과적으로 나타내기 때문입니다. 열린 **IndexReader** 내에서는 문서 목록을 변경할 수 없습니다. 그러나 다른/새 **IndexReader** 인스턴스는 다른 문서 집합(다른 인덱스에서 또는 인덱스가 변경되었기 때문에 간단히) 작업하므로 캐시된 **DocIdSet**을 다시 계산해야 합니다.

또한 **Hibernate Search**는 이러한 캐싱 측면에도 도움이 됩니다. 기본적으로 **@FullTextFilterDef**의 캐시 플래그는 **FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS** 로 설정되어 필터 인스턴스를 자동으로 캐시하고 **CachingWrapperFilter**의 **Hibernate** 특정 구현을 중심으로 지정된 필터를 래핑합니다. **Lucene**의 이 클래스 버전에 비해 **softReferences**는 하드 참조 수와 함께 사용됩니다(필터 캐시에 대한 토론 참조). 하드 참조 수는 **hibernate.search.filter.cache_docidresults.size(기본값: 5)**를 사용하여 조정할 수 있습니다. 래핑 작업은 **@FullTextFilterDef.cache** 매개 변수를 사용하여 제어할 수 있습니다. 이 매개변수에는 세 가지 다른 값이 있습니다.

현재의	정의
FilterCacheModeType.NONE	필터 인스턴스가 없으며 Hibernate Search에 의해 캐시되지 않습니다. 모든 필터 호출에 대해 새 필터 인스턴스가 생성됩니다. 이 설정은 신속한 데이터 집합 또는 메모리 제한 환경에 유용할 수 있습니다.
FilterCacheModeType.INSTANCE_ONLY	필터 인스턴스가 캐시되고 Filter.getDocIdSet() 호출에서 재사용됩니다. DocIdSet 결과가 캐시되지 않습니다. 이 설정은 필터가 고유한 특정 캐싱 메커니즘을 사용하거나 필터 결과가 애플리케이션별 이벤트를 통해 DocIdSet 캐싱을 불필요하게 만들기 때문에 동적으로 변경될 때 유용합니다.
FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS	필터 인스턴스와 DocIdSet 결과 모두 캐시됩니다. 이는 기본값입니다.

필터를 다음과 같은 상황에서 캐시해야 합니다.

- 시스템은 대상 엔터티 인덱스를 자주 업데이트하지 않습니다(즉, **IndexReader**는 많이 재사용됨)

- 필터의 **DocIdSet**은 계산에 비용이 많이 듭니다 (쿼터를 실행하는 데 소요되는 시간보다)

7.5.2.6. 공유된 환경에서 필터 사용

분할된 환경에서는 사용 가능한 **shard**의 하위 집합에 대한 쿼리를 실행할 수 있습니다. 이 작업은 다음 두 단계로 수행할 수 있습니다.

인덱스 공유의 하위 집합 쿼리

1. 필터 구성에 따라 **IndexManager**의 하위 집합을 선택하는 분할 전략을 생성합니다.

2. 쿼리 시 필터를 활성화합니다.

예제: 인덱스 공유의 하위 집합 쿼리

이 예제에서는 고객 필터가 활성화된 경우 특정 고객 **shard**에 대해 쿼리가 실행됩니다.

```
public class CustomerShardingStrategy implements IndexShardingStrategy {

    // stored IndexManagers in an array indexed by customerID
    private IndexManager[] indexManagers;

    public void initialize(Properties properties, IndexManager[] indexManagers) {
      this.indexManagers = indexManagers;
    }

    public IndexManager[] getIndexManagersForAllShards() {
      return indexManagers;
    }

    public IndexManager getIndexManagerForAddition(
        Class<?> entity, Serializable id, String idInString, Document document) {
      Integer customerID =
Integer.parseInt(document.getFieldable("customerID").stringValue());
      return indexManagers[customerID];
    }

    public IndexManager[] getIndexManagersForDeletion(
        Class<?> entity, Serializable id, String idInString) {
      return getIndexManagersForAllShards();
    }
```

```
 /**
 * Optimization; don't search ALL shards and union the results; in this case, we
 * can be certain that all the data for a particular customer Filter is in a single
 * shard; simply return that shard by customerID.
 */
public IndexManager[] getIndexManagersForQuery(
    FullTextFilterImplementor[] filters) {
  FullTextFilter filter = getCustomerFilter(filters, "customer");
  if (filter == null) {
    return getIndexManagersForAllShards();
  }
  else {
    return new IndexManager[] { indexManagers[Integer.parseInt(
      filter.getParameter("customerID").toString())] };
  }
}

private FullTextFilter getCustomerFilter(FullTextFilterImplementor[] filters, String name) {
  for (FullTextFilterImplementor filter: filters) {
    if (filter.getName().equals(name)) return filter;
  }
  return null;
}
}
```

이 예에서 **customer** 라는 필터가 있으면 이 고객 전용 **shard**만 쿼리됩니다. 그렇지 않으면 모든 **shard**가 반환됩니다. 지정된 분할 전략은 하나 이상의 필터에 반응하고 해당 매개변수에 따라 달라집니다.

두 번째 단계는 쿼리 시간에 필터를 활성화하는 것입니다. 필터는 쿼리 후 **Lucene** 결과를 필터링하는 일반 필터(에 정의된 대로)일 수 있지만 **sharding** 전략에만 전달될 특수 필터를 사용할 수 있습니다(및 무시됨).

이 기능을 사용하려면 필터를 선언할 때 **ShardSensitiveOnlyFilter** 클래스를 지정합니다.

```
@Indexed
@FullTextFilterDef(name="customer", impl=ShardSensitiveOnlyFilter.class)
public class Customer {
  ...
}

FullTextQuery query = ftEm.createFullTextQuery(luceneQuery, Customer.class);
query.enableFulltextFilter("customer").setParameter("CustomerID", 5);
@SuppressWarnings("unchecked")
List<Customer> results = query.getResultList();
```

ShardSensitiveOnlyFilter를 사용하면 **Lucene** 필터를 구현할 필요가 없습니다. 이러한 필터에 대응하는 필터 및 분할 전략을 사용하면 **sharded** 환경에서 쿼리 속도를 높일 수 있습니다.

7.5.3. Faceting

facet 검색은 쿼리 결과를 여러 카테고리로 나눌 수 있는 기술입니다. 이 분류에는 각 범주에 대한 적중 횟수 계산 및 이러한 **facet**(통합)를 기반으로 검색 결과를 추가로 제한하는 기능이 포함됩니다. 아래 예제에서는 **faceting** 예제를 보여줍니다. 검색 결과는 페이지의 주요 부분에 표시되는 **15**개의 조회 수입니다. 하지만 왼쪽에 있는 네비게이션 바에서는 *프로그래밍*,*컴퓨터 과학*,*데이터베이스*,*소프트웨어*,*웹 개발*, *네트워킹* 및 *홈 컴퓨팅을 기준으로 분류된* ***Computers & Internet*** *카테고리를 보여 줍니다.* 이러한 각 하위 범주의 경우 주요 검색 기준에 맞는 서적 수가 표시되며 해당 하위 범주에 속합니다. 컴퓨터 *및 인터넷 부문의* 이 부문은 구체적인 검색 **facet** 중 하나입니다. 또 다른 하나는 평균 고객 검토입니다.

facet 검색은 쿼리 결과를 카테고리로 나눕니다. 범주화에는 각 범주에 대한 적중 횟수 계산이 포함되며 이러한 **facet**(통계)를 기준으로 검색 결과를 추가로 제한합니다. 다음 예제는 **faceting** 검색 결과를 기본 페이지에 **15**회 표시합니다.

왼쪽 네비게이션 바에는 범주와 하위 범주가 표시됩니다. 이러한 각 하위 범주의 경우 도서 수는 주요 검색 기준과 일치하며 해당 하위 범주에 속합니다. 컴퓨터 및 인터넷 부문의 이 부문은 구체적인 검색 **facet** 중 하나입니다. 또 다른 예로는 평균 고객 검토가 있습니다.

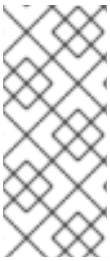
예제: Amazon에서 Hibernate Search 검색

Hibernate Search에서 **QueryBuilder** 및 **FullTextQuery** 클래스는 **faceting API**의 진입점입니다. 전자는 **faceting** 요청을 만들고 후자는 **FacetManager**에 액세스합니다. **FacetManager**는 쿼리에 **faceting** 요청을 적용하고 기존 쿼리에 추가된 **facet**를 선택하여 검색 결과를 구체화합니다. 예에서는 아래 예와 같이 **Cd** 엔터티를 사용합니다.

예제: 엔터티 Cd

```
@Indexed
public class Cd {

   private int id;

   @Fields( {
      @Field,
      @Field(name = "name_un_analyzed", analyze = Analyze.NO)
```

```
    })
    private String name;

    @Field(analyze = Analyze.NO)
    @NumericField
    private int price;

    Field(analyze = Analyze.NO)
    @DateBridge(resolution = Resolution.YEAR)
    private Date releaseYear;

    @Field(analyze = Analyze.NO)
    private String label;


// setter/getter
...
```



참고

Hibernate Search 5.2 이전에는 @Facet 주석을 명시적으로 사용할 필요가 없었습니다. Hibernate Search 5.2에서는 Lucene의 기본 faceting API를 사용하려면 필수가 되었습니다.

7.5.3.1. Faceting 요청 생성

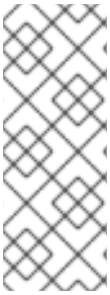
facet 검색을 위한 첫 번째 단계는 FacetingRequest를 생성하는 것입니다. 현재 두 가지 유형의 페이징 요청이 지원됩니다. 첫 번째 유형을 *개별 faceting* 및 두 번째 유형 *범위 faceting* 요청이라고 합니다. 개별 faceting 요청의 경우 facet(분산)할 인덱스 필드와 적용할 옵션을 지정합니다. 개별 다면 요청의 예는 다음 예에서 확인할 수 있습니다.

예제: 개별 Faceting 요청 생성

```
QueryBuilder builder = fullTextSession.getSearchFactory()
    .buildQueryBuilder()
        .forEntity( Cd.class )
            .get();
FacetingRequest labelFacetingRequest = builder.facet()
    .name( "labelFaceting" )
    .onField( "label")
    .discrete()
    .orderedBy( FacetSortOrder.COUNT_DESC )
```

```
    .includeZeroCounts( false )
    .maxFacetCount( 1 )
    .createFacetingRequest();
```

이 faceting 요청을 실행하면 색인화된 필드 레이블 의 각 개별 값에 대해 Facet 인스턴스가 생성됩니다. Facet 인스턴스는 원래 쿼리 결과 내에서 이 특정 필드 값이 얼마나 자주 발생하는지를 포함하여 실제 필드 값을 기록합니다. ordersBy에는 registriesCounts 및 maxFacetCount가 포함되며 모든 faceting 요청에 적용할 수 있는 선택적 매개변수입니다. orderedBy는 생성된 facet가 반환되는 순서를 지정할 수 있습니다. 기본값은 FacetSortOrder.COUNT_DESC 이지만 필드 값을 기준으로 정렬하거나 범위를 지정한 순서로 정렬할 수도 있습니다. include를 포함한 facet는 개수가 0인 facet가 결과에 포함될지 여부를 결정하고 maxFacetCount는 반환된 facet의 최대 양을 제한할 수 있습니다.



참고

지금은 faceting을 적용하기 위해 색인화된 필드를 충족해야 하는 몇 가지 전제 조건이 있습니다. 색인화된 속성은 유형 String, Date 또는 subtype of Number and null 값이어야 합니다. 또한 속성은 Analyze.NO 로 인덱싱되어야 하며 숫자 속성 @NumericField를 지정해야 합니다.

범위 턴팅 요청 생성은 우리가 직면하고 있는 필드 값에 대한 범위를 지정해야 한다는 점을 제외하고 매우 유사합니다. 범위 다면 요청은 세 가지 가격 범위가 지정된 아래에서 확인할 수 있습니다. 아래 와 위의 내용은 한 번만 지정할 수 있지만 원하는 만큼 from -부터 범위를 지정할 수 있습니다. 각 범위 경계에 대해 범위에 포함되었는지 여부를 excludeLimit를 통해 지정할 수도 있습니다.

예제: 범위 숨기기 요청 생성

```
QueryBuilder builder = fullTextSession.getSearchFactory()
    .buildQueryBuilder()
        .forEntity( Cd.class )
            .get();
FacetingRequest priceFacetingRequest = builder.facet()
    .name( "priceFaceting" )
    .onField( "price" )
    .range()
    .below( 1000 )
    .from( 1001 ).to( 1500 )
    .above( 1500 ).excludeLimit()
    .createFacetingRequest();
```

7.5.3.2. Faceting 요청 적용

faceting 요청은 **FullTextQuery** 클래스를 통해 검색할 수 있는 **FacetManager** 클래스를 통해 쿼리에 적용됩니다.

faceting 요청 이름을 지정하는 **getFacets()**를 통해 원하는 만큼의 **faceting** 요청을 활성화하고 나중에 검색할 수 있습니다. 또한 이름을 지정하여 **faceting** 요청을 비활성화할 수 있는 **disableFaceting()** 메서드도 있습니다.

faceting 요청은 **FullTextQuery**를 통해 검색할 수 있는 **FacetManager**를 사용하여 쿼리에 적용할 수 있습니다.

예제: Faceting 요청 적용

```
// create a fulltext query
Query luceneQuery = builder.all().createQuery(); // match all query
FullTextQuery fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery, Cd.class );

// retrieve facet manager and apply faceting request
FacetManager facetManager = fullTextQuery.getFacetManager();
facetManager.enableFaceting( priceFacetingRequest );

// get the list of Cds
List<Cd> cds = fullTextQuery.list();
...

// retrieve the faceting results
List<Facet> facets = facetManager.getFacets( "priceFaceting" );
...
```

getFacets() 및 **faceting** 요청 이름을 지정하여 여러 다면 지정 요청을 검색할 수 있습니다.

disableFaceting() 메서드는 이름을 지정하여 **faceting** 요청을 비활성화합니다.

7.5.3.3. 쿼리 결과 제한

마지막으로 반환된 **Facets** 중 하나를 **"drill-down"** 기능을 구현하기 위해 원래 쿼리에 대한 추가 기준으로 적용할 수 있습니다. 이 목적을 위해 **FacetSelection**을 사용할 수 있습니다. **FacetSelection**은

FacetManager를 통해 사용할 수 있으며 쿼리 기준(selectFacets)으로 facet를 선택하고, facet 제한(deselectFacets)을 제거하고 모든 facet 제한(clearSelectedFacets)을 제거하고 현재 선택된 facet(getSelectedFacets)를 검색합니다. 다음 코드 조각은 예를 보여줍니다.

```
// create a fulltext query
Query luceneQuery = builder.all().createQuery(); // match all query
FullTextQuery fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery, clazz );

// retrieve facet manager and apply faceting request
FacetManager facetManager = fullTextQuery.getFacetManager();
facetManager.enableFaceting( priceFacetingRequest );

// get the list of Cd
List<Cd> cds = fullTextQuery.list();
assertTrue(cds.size() == 10);

// retrieve the faceting results
List<Facet> facets = facetManager.getFacets( "priceFaceting" );
assertTrue(facets.get(0).getCount() == 2)

// apply first facet as additional search criteria
facetManager.getFacetGroup( "priceFaceting" ).selectFacets( facets.get( 0 ) );

// re-execute the query
cds = fullTextQuery.list();
assertTrue(cds.size() == 2);
```

7.5.4. 쿼리 프로세스 최적화

쿼리 성능은 몇 가지 기준에 따라 다릅니다.

- Lucene 쿼리.
- 로드된 오브젝트 수: 페이지 번호(항상) 또는 인덱스 프로젝션(필요한 경우)을 사용합니다.
- Hibernate Search가 Lucene 리더와 상호 작용하는 방식: 적절한 리더 전략을 정의합니다.
- 인덱스에서 자주 추출된 값을 캐싱합니다. 계산 색인 값 참조: FieldCache 자세한 내용을 보려면 다음을 수행하십시오.

7.5.4.1. 캐싱 인덱스 값: FieldCache

Lucene 인덱스의 주요 기능은 쿼리와 일치하는지 식별하는 것입니다. 쿼리를 수행한 후 유용한 정보

를 추출하려면 결과를 분석해야 합니다. Hibernate Search는 일반적으로 클래스 유형과 기본 키를 추출해야 합니다.

인덱스에서 필요한 값을 추출하면 성능 비용이 발생하기 때문에 경우에 따라 매우 낮고 눈에 띄지 않을 수 있지만 다른 경우에는 캐싱에 적합한 후보가 될 수 있습니다.

요구 사항은 쿼리 컨텍스트 또는 기타 수단에서 유추할 수 있으므로 클래스 유형이 필요하지 않으므로 사용 중인 예상 유형에 따라 달라집니다.

@CacheFromIndex 주석을 사용하면 Hibernate Search에 필요한 기본 metadata 필드의 다양한 유형의 캐싱을 실험할 수 있습니다.

```
import static org.hibernate.search.annotations.FieldCacheType.CLASS;
import static org.hibernate.search.annotations.FieldCacheType.ID;

@Indexed
@CacheFromIndex( { CLASS, ID } )
public class Essay {
   ...
```

이 주석을 사용하여 클래스 유형 및 ID를 캐시할 수 있습니다.

- CLASS: Hibernate Search는 Lucene FieldCache를 사용하여 인덱스에서 추출된 클래스 유형의 성능을 향상시킵니다.

  이 값은 기본적으로 활성화되며 @CacheFromIndex 주석을 지정하지 않으면 Hibernate Search가 적용되는 작업입니다.

- ID: 기본 식별자를 추출하면 캐시가 사용됩니다. 이는 최상의 수행 쿼리를 제공할 가능성이 높지만 더 많은 메모리를 소비하므로 성능이 저하될 수 있습니다.

참고

웜업 후 성능 및 메모리 사용 영향을 측정합니다(일부 쿼리 실행). 필드 캐시를 활성화하여 성능이 향상될 수 있지만 항상 그런 것은 아닙니다.

FieldCache를 사용하면 고려해야 할 두 가지 단점이 있습니다.

- 메모리 사용량: 이러한 캐시는 상당히 메모리가 많이 소모될 수 있습니다. 일반적으로 CLASS 캐시는 ID 캐시보다 요구 사항이 낮습니다.

- 인덱스 준비: 필드 캐시를 사용할 때 새 인덱스의 첫 번째 쿼리 또는 세그먼트가 활성화된 캐싱이 없는 경우보다 느립니다.

일부 쿼리에서는 클래스 유형이 전혀 필요하지 않습니다. 이 경우 CLASS 필드 캐시를 활성화해도 사용되지 않을 수 있습니다. 예를 들어 단일 클래스를 대상으로 하는 경우 반환된 모든 값은 해당 유형이 됩니다(각 쿼리 실행 시 평가됨).

ID FieldCache를 사용하려면 대상 엔터티의 ID가 TwoWayFieldBridge(모든 빌드 브리지)를 사용해야 하며 특정 쿼리에 로드되는 모든 유형은 id에 필드 이름을 사용해야 하며 동일한 유형의 ID가 있어야 합니다(각 쿼리 실행 시 평가됨).

7.6. 수동 인덱스 변경

Hibernate Core가 데이터베이스에 변경 사항을 적용함에 따라 Hibernate Search는 이러한 변경 사항을 탐지하고 인덱스를 자동으로 업데이트합니다(EventListener가 비활성화되지 않은 경우). 백업이 복원되거나 데이터가 다른 영향을 받는 경우와 같이 Hibernate를 사용하지 않고 데이터베이스를 변경할 수 있습니다. 이러한 경우 Hibernate Search는 Manual Index API를 노출하여 인덱스에서 단일 엔터티를 명시적으로 업데이트하거나 제거하거나, 전체 데이터베이스의 인덱스를 다시 빌드하거나, 특정 유형에 대한 모든 참조를 제거합니다.

이러한 모든 메서드는 Lucene 인덱스에만 영향을 미치며 변경 사항이 데이터베이스에 적용되지 않습니다.

7.6.1. 인덱스에 인스턴스 추가

FullTextSession.index(T 엔터티) 를 사용하여 특정 오브젝트 인스턴스를 인덱스에 직접 추가하거나 업데이트할 수 있습니다. 이 엔터티가 이미 인덱싱된 경우 인덱스가 업데이트됩니다. 인덱스 변경 사항은 트랜잭션 커밋에만 적용됩니다.

FullTextSession.index(T 엔터티) 를 사용하여 오브젝트 또는 인스턴스를 인덱스에 직접 추가합니다. 엔터티가 인덱싱되면 인덱스가 업데이트됩니다. Infinispan 쿼리는 트랜잭션 커밋 중에 인덱스에 변경 사항을 적용합니다.

예제: Full TextSession.index(T entity)를 사용하여 엔터티 인덱싱

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
Object customer = fullTextSession.load( Customer.class, 8 );
fullTextSession.index(customer);
tx.commit(); //index only updated at commit time
```

유형에 대한 모든 인스턴스를 추가하거나 인덱싱된 모든 유형에 대해 권장되는 접근법은 **MassIndexer**를 사용하는 것입니다. 자세한 내용은 를 참조하십시오.

amountIndexer를 사용하여 유형 또는 인덱싱된 모든 유형에 대한 모든 인스턴스를 추가합니다. 자세한 내용은 **MassIndexer** 사용을 참조하십시오.

7.6.2. 인덱스에서 인스턴스 삭제

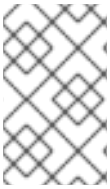
데이터베이스에서 물리적으로 제거하지 않고도 특정 유형의 엔터티 또는 모든 엔터티를 **Lucene** 인덱스에서 제거할 수 있습니다. 이 작업의 이름은 제거이며 **Full TextSession** 을 통해서도 수행됩니다.

삭제 작업을 사용하면 데이터베이스에서 물리적으로 제거하지 않고 **Lucene** 인덱스에서 단일 엔터티 또는 지정된 유형의 모든 엔터티를 제거할 수 있습니다. 이 작업은 **Full TextSession**을 사용하여 수행합니다.

예제: 인덱스에서 엔터티의 특정 인스턴스 제거

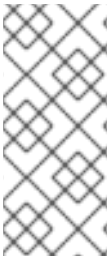
```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
for (Customer customer : customers) {
fullTextSession.purgeAll( Customer.class );
//optionally optimize the index
//fullTextSession.getSearchFactory().optimize( Customer.class );
tx.commit(); //index is updated at commit time
```

이러한 작업을 수행한 후 인덱스를 최적화하는 것이 좋습니다.



참고

메서드 인덱스, 제거 및 제거All은 FullTextEntityManager에서도 사용할 수 있습니다.



참고

모든 수동 인덱싱 메서드(index, purge, purgeAll)는 데이터베이스가 아닌 인덱스에만 영향을 미치고 트랜잭션이 성공적으로 커밋되거나 flushToIndexes를 사용할 때까지 적용되지 않습니다.

7.6.3. 인덱스 다시 빌드

엔터티 매핑을 인덱스로 변경하는 경우 전체 인덱스를 업데이트해야 합니다. 예를 들어 다른 분석기를 사용하여 기존 필드를 인덱싱하기로 결정한 경우 영향을 받는 유형에 대한 인덱스를 다시 빌드해야 합니다. 데이터베이스가 교체된 경우에도(기존 시스템에서 가져온 백업에서 복원된 것처럼) 기존 데이터에서 인덱스를 다시 빌드할 수 있습니다. Hibernate Search는 다음과 같은 두 가지 주요 전략을 제공합니다.

인덱서에서 엔터티 매핑을 변경하려면 전체 인덱스를 업데이트해야 할 수 있습니다. 예를 들어 다른 분석기를 사용하여 기존 필드를 인덱싱하려면 영향을 받는 유형에 대해 인덱스를 다시 빌드해야 합니다.

또한 백업에서 복원하거나 레거시 시스템에서 가져와서 데이터베이스를 교체하는 경우 기존 데이터에서 인덱스를 다시 빌드해야 합니다. Infinispan 쿼리는 다음 두 가지 주요 전략을 제공합니다.

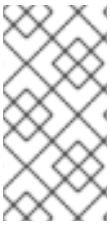
- Full TextSession.flushToIndexes() 를 주기적으로 사용하는 동안 모든 엔터티에서 Full TextSession.index() 를 사용합니다.
- MassIndexer 사용.

7.6.3.1. Using flushToIndexes()

이 전략은 기존 인덱스를 제거한 다음 FullTextSession.purgeAll() 및 FullTextSession.index() 를 사용하여 인덱스에 모든 엔터티를 다시 추가하는 것으로 구성되지만 몇 가지 메모리 및 효율성 제약 조건이 있습니다. 최대 효율성을 위해 Hibernate Search 인덱스 작업을 배치하고 커밋 시 실행합니다. 트랜잭션 커밋까지 모든 문서가 큐에 보관되므로 많은 데이터를 인덱싱해야 합니다. 큐를 정기적으로 비우지 않는 경우 OutOfMemoryException 에 직면할 수 있습니다. 이렇게 하려면 fullTextSession.flushToIndexes() 를 사용합니다. fulltextSession.flushToIndexes() 를 호출할 때마다(또는 트랜잭션이 커밋된 경우) 배치 대기열이 처리되고 모든 인덱스 변경 사항을 적용합니다. 플러시되고 나면 변경 사항을 롤백할 수 없습니다.

예제: index() 및 flushToIndexes()를 사용한 인덱스 재구축

```
fullTextSession.setFlushMode(FlushMode.MANUAL);
fullTextSession.setCacheMode(CacheMode.IGNORE);
transaction = fullTextSession.beginTransaction();
//Scrollable results will avoid loading too many objects in memory
ScrollableResults results = fullTextSession.createCriteria( Email.class )
    .setFetchSize(BATCH_SIZE)
    .scroll( ScrollMode.FORWARD_ONLY );
int index = 0;
while( results.next() ) {
    index++;
    fullTextSession.index( results.get(0) ); //index each element
    if (index % BATCH_SIZE == 0) {
        fullTextSession.flushToIndexes(); //apply changes to indexes
        fullTextSession.clear(); //free memory since the queue is processed
    }
}
transaction.commit();
```



참고

Hibernate.search.default.worker.batch_size 는 더 나은 제어를 제공하는 명시적 API 대신 더 이상 사용되지 않음

애플리케이션이 메모리가 부족해지는 것을 보장하는 배치 크기를 사용하십시오. 데이터베이스에서 배치 크기 오브젝트가 더 빨리 가져오지만 더 많은 메모리가 필요합니다.

7.6.3.2. MassIndexer 사용

Hibernate Search의 MassIndexer는 여러 병렬 스레드를 사용하여 인덱스를 다시 빌드합니다. 선택적으로 다시 로드해야 하는 엔터티를 선택하거나 모든 엔터티를 다시 인덱스하도록 할 수 있습니다. 이 방법은 최상의 성능을 위해 최적화되지만 유지 관리 모드로 애플리케이션을 설정해야 합니다. MassIndexer가 사용 중인 경우 인덱스 쿼리는 권장되지 않습니다.

예제: MassIndexer를 사용하여 인덱스 다시 빌드

```
fullTextSession.createIndexer().startAndWait();
```

이렇게 하면 인덱스를 다시 빌드하여 삭제한 다음 데이터베이스에서 모든 엔터티를 다시 로드합니다. 사용이 간단하지만, 프로세스 속도를 높이기 위해 일부 조정을 권장합니다.

주의

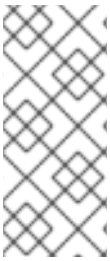
MassIndexer가 진행되는 동안 인덱스의 내용이 정의되지 않습니다. **MassIndexer**가 작업하는 동안 쿼리를 수행하면 일부 결과가 누락될 가능성이 큽니다.

예제: tuned MassIndexer 사용

```
fullTextSession
 .createIndexer( User.class )
 .batchSizeToLoadObjects( 25 )
 .cacheMode( CacheMode.NORMAL )
 .threadsToLoadObjects( 12 )
 .idFetchSize( 150 )
 .progressMonitor( monitor ) //a MassIndexerProgressMonitor implementation
 .startAndWait();
```

그러면 모든 사용자 인스턴스(및 하위 유형)의 인덱스가 다시 빌드되고 쿼리당 **25**개 오브젝트의 배치를 사용하여 **User** 인스턴스를 로드할 **12**개의 병렬 스레드가 생성됩니다. 이 **12**개 스레드는 **Lucene** 문서를 출력하기 위해 인덱싱된 임베디드 관계 및 사용자 정의 **Field Bridges** 또는 **ClassBridges** 를 처리해야 합니다. 스레드는 변환 프로세스 중에 추가 속성의 지연 로드를 트리거합니다. 이로 인해 병렬로 작동하는 많은 스레드가 필요합니다. 실제 인덱스 쓰기에 사용되는 스레드 수는 각 인덱스의 백엔드 구성으로 정의됩니다.

대부분의 재지급 상황에서 캐시가 사용되지 않는 추가 오버헤드가 발생하므로 **cache Mode.IGNORE**(기본값)를 **CacheMode.IGNORE**(기본값)로 두는 것이 좋습니다. 기본 엔터티가 인덱스에 포함된 열거와 유사한 데이터와 관련된 경우 데이터를 향상시킬 수 있으므로 다른 **CacheMode** 를 활성화하는 것이 유용할 수 있습니다.



참고

최적의 성능을 얻기 위해 이상적인 스레드 수는 전체 아키텍처, 데이터베이스 설계 및 데이터 값에 따라 달라집니다. 모든 내부 스레드 그룹에는 의미 있는 이름이 있으므로 스레드 덤프를 포함하여 대부분의 진단 도구로 쉽게 식별해야 합니다.

참고

MassIndexer는 트랜잭션을 인식하지 못하므로 나중에 한 번 또는 커밋을 시작할 필요가 없습니다. 트랜잭션이 아니므로 사용자가 처리 중에 시스템을 사용하도록 두지 않는 것이 좋습니다. 예기치 않은 사용자가 결과를 찾을 수 없고 시스템 부하가 너무 높을 수 있습니다.

인덱싱 시간과 메모리 사용량에 영향을 주는 기타 매개변수는 다음과 같습니다.

- **hibernate.search.[default|<indexname>].exclusive_index_use**
- **hibernate.search.[default|<indexname>].indexwriter.max_buffered_docs**
- **hibernate.search.[default|<indexname>].indexwriter.max_merge_docs**
- **hibernate.search.[default|<indexname>].indexwriter.merge_factor**
- **hibernate.search.[default|<indexname>].indexwriter.merge_min_size**
- **hibernate.search.[default|<indexname>].indexwriter.merge_max_size**
- **hibernate.search.[default|<indexname>].indexwriter.merge_max_optimize_size**
- **hibernate.search.[default|<indexname>].indexwriter.merge_calibrate_by_deletes**
- **hibernate.search.[default|<indexname>].indexwriter.ram_buffer_size**

- hibernate.search.[default|<indexname>].indexwriter.term_index_interval

이전 버전에는 max_field_length 도 있었지만 Lucene에서 제거되었습니다. LimitTokenCountAnalyzer 를 사용하여 유사한 효과를 얻을 수 있습니다.

all .indexwriter 매개변수는 Lucene 고유하며 Hibernate Search는 이러한 매개변수를 전달합니다.

MassIndexer는 로드할 기본 키를 반복하기 위해 앞으로만 스크롤 가능한 결과를 사용하지만 MySQL의 JDBC 드라이버는 메모리의 모든 값을 로드합니다. 이 "최적화"를 방지하려면 idFetchSize 를 Integer.MIN_VALUE 로 설정합니다.

7.7. 인덱스 최적화

경우에 따라 Lucene 인덱스를 최적화해야 합니다. 이 프로세스는 기본적으로 조각 모음입니다. 최적화가 트리거될 때까지 Lucene은 삭제된 문서만 표시하므로 실제가 적용되지 않습니다. 최적화 프로세스 중에 Lucene Directory의 파일 수에도 영향을 주는 삭제가 적용됩니다.

Lucene 인덱스를 최적화하면 검색 속도가 빨라지지만 색인화(업데이트) 성능에는 영향을 미치지 않습니다. 최적화하는 동안 검색을 수행할 수 있지만 속도가 느려질 가능성이 높습니다. 모든 인덱스 업데이트가 중지됩니다. 최적화를 예약하는 것이 좋습니다.

Lucene 인덱스를 최적화하면 검색 속도가 빨라지지만 인덱스 업데이트 성능에는 영향을 미치지 않습니다. 최적화 프로세스 중에 검색을 수행할 수 있지만 예상보다 느립니다. 모든 인덱스 업데이트는 최적화 중에 보류 상태로 유지됩니다. 따라서 최적화를 예약하는 것이 좋습니다.

- 유휴 시스템 또는 검색이 가장 빈번한 경우.
- 많은 인덱스 수정을 적용한 후.

MassIndexer는 처리 시작 및 종료 시 기본적으로 인덱스를 최적화합니다. 이 기본 동작을 변경하려면 MassIndexer.optimizeAfterPurge 및 MassIndexer.optimizeOnFinish 를 사용합니다. 자세한 내용은 MassIndexer 사용을 참조하십시오.

7.7.1. 자동 최적화

다음 중 하나를 수행하면 **Hibernate Search**에서 인덱스를 자동으로 최적화할 수 있습니다.

다음 후 **Infinispan** 쿼리에서 인덱스를 자동으로 최적화합니다.

- 특정 작업 수(세부 또는 삭제)입니다.
- 특정 금액의 트랜잭션.

자동 인덱스 최적화 구성은 전역 또는 인덱스별로 정의할 수 있습니다.

예제: 자동 최적화 매개변수 정의

```
hibernate.search.default.optimizer.operation_limit.max = 1000
hibernate.search.default.optimizer.transaction_limit.max = 100
hibernate.search.Animal.optimizer.transaction_limit.max = 50
```

다음과 같은 즉시 최적화가 트리거됩니다.

- 추가 및 삭제 수는 **1000** 개에 도달합니다.
- 트랜잭션 수 **50** (**hibernate.search.Animal.optimizer.transaction_limit.max**는 **hibernate.search.default.optimizer.transaction_limit.max**보다 우선 순위)에 도달합니다.

이러한 매개변수가 정의되어 있지 않으면 최적화가 자동으로 처리되지 않습니다.

OptimizerStrategy의 기본 구현은 **org.hibernate.search.store.optimization.OptimizerStrategy** 를 구현 의 정규화된 이름으로 설정하여 재정의할 수 있습니다. 이 구현에서는 인터페이스를 구현해야 하며 공용 클래스가 아니며 인수를 사용하지 않는 공용 생성자가 있어야 합니다.

예제: 사용자 지정 OptimizerStrategy 로드

```
hibernate.search.default.optimizer.implementation =
com.acme.worlddomination.SmartOptimizer
hibernate.search.default.optimizer.SomeOption = CustomConfigurationValue
hibernate.search.humans.optimizer.implementation = default
```

키워드 기본값 을 사용하여 Hibernate Search 기본 구현을 선택할 수 있습니다 .optimizer 키 구분 기호 이후의 모든 속성은 구현의 초기화 방법으로 전달됩니다.

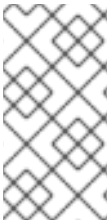
7.7.2. 수동 최적화

Hibernate Search에서 SearchFactory를 통해 Lucene 인덱스를 프로그래밍 방식으로 최적화할 수 있습니다.

예제: 프로그래밍 방식 인덱스 최적화

```
FullTextSession fullTextSession = Search.getFullTextSession(regularSession);
SearchFactory searchFactory = fullTextSession.getSearchFactory();

searchFactory.optimize(Order.class);
// or
searchFactory.optimize();
```

첫 번째 예는 주문을 보유하는 Lucene 인덱스를 최적화하고 두 번째 예에서는 모든 인덱스를 최적화합니다.



참고

searchFactory.optimize() 는 자카르타 메시징 백엔드에 영향을 미치지 않습니다. 마스터 노드에서 최적화 작업을 적용해야 합니다.

searchFactory.optimize() 는 자카르타 메시징 백엔드에 영향을 주지 않기 때문에 마스터 노드에 적용

됩니다.

7.7.3. 최적화 조정

Apache Lucene에는 최적화를 수행하는 방법에 영향을 주는 몇 가지 매개 변수가 있습니다. Hibernate Search는 이러한 매개 변수를 노출합니다.

추가 인덱스 최적화 매개 변수는 다음과 같습니다.

- hibernate.search.[default|<indexname>].indexwriter.max_buffered_docs
- hibernate.search.[default|<indexname>].indexwriter.max_merge_docs
- hibernate.search.[default|<indexname>].indexwriter.merge_factor
- hibernate.search.[default|<indexname>].indexwriter.ram_buffer_size
- hibernate.search.[default|<indexname>].indexwriter.term_index_interval

7.8. 고급 기능

7.8.1. SearchFactory에 액세스

SearchFactory 개체는 Hibernate 검색을 위한 기본 Lucene 리소스를 추적합니다. 기본적으로 Lucene에 액세스할 수 있는 편리한 방법입니다. SearchFactory 는 FullTextSession에서 액세스할 수 있습니다.

예제: SearchFactory에 액세스

```
FullTextSession fullTextSession = Search.getFullTextSession(regularSession);
SearchFactory searchFactory = fullTextSession.getSearchFactory();
```

7.8.2. IndexReader 사용

Lucene의 쿼리는 IndexReader에서 실행됩니다. Hibernate Search는 성능을 극대화하기 위해 인덱스 리더를 캐시하거나 I/O 작업을 최소화하는 업데이트된 IndexReader를 검색하는 다른 효율적인 전략을 제공할 수 있습니다. 코드는 이러한 캐시 리소스에 액세스할 수 있지만 몇 가지 요구 사항이 있습니다.

예제: IndexReader 액세스

```
IndexReader reader = searchFactory.getIndexReaderAccessor().open(Order.class);
try {
  //perform read-only operations on the reader
}
finally {
  searchFactory.getIndexReaderAccessor().close(reader);
}
```

이 예에서 SearchFactory는 이 엔터티를 쿼리하는 데 필요한 인덱스를 결정합니다(sharding 전략 고려). 각 인덱스에 구성된 ReaderProvider를 사용하면 관련된 모든 인덱스 위에 복합 IndexReader 가 반환됩니다. 이 IndexReader는 여러 클라이언트 간에 공유되므로 다음 규칙을 준수해야 합니다.

- indexReader.close()를 호출하지 않고 필요한 경우 readerProvider.closeReader(reader)를 사용합니다.
- 수정 작업에 이 IndexReader를 사용하지 마십시오(읽기 전용 IndexReader이므로 모든 시도는 예외가 발생합니다).

이러한 규칙 외에도 특히 네이티브 Lucene 쿼리를 수행하기 위해 IndexReader를 자유롭게 사용할 수 있습니다. shared IndexReaders를 사용하면 파일 시스템에서 직접 에서 직접 쿼리를 열 때보다 대부분의 쿼리의 효율성이 향상됩니다.

메서드에 대한 대안으로 open(Class… type) open(String… indexNames: 하나 이상의 인덱스 이름을 전달할 수 있습니다. 이 전략을 사용하면 분할이 사용되는 경우 인덱싱된 유형에 대한 인덱스의 하위 집합을 선택할 수도 있습니다.

예제: 인덱스 이름으로 IndexReader에 액세스

```
IndexReader reader = searchFactory.getIndexReaderAccessor().open("Products.1",
"Products.3");
```

7.8.3. Lucene 디렉토리 액세스

Directory는 인덱스 스토리지를 표현하기 위해 Lucene이 사용하는 가장 일반적인 추상화입니다. Hibernate Search는 Lucene Directory와 직접 상호 작용하지 않지만 IndexManager를 통해 이러한 상호 작용을 추상화합니다. 인덱스가 디렉터리가 구현하지 않아도 됩니다.

인덱스가 Directory로 표시되고 액세스해야 하는 경우 IndexManager를 통해 Directory에 대한 참조를 얻을 수 있습니다. IndexManager를 DirectoryBasedIndexManager에 포착한 다음 getDirectoryProvider().getDirectory() 를 사용하여 기본 디렉터리에 대한 참조를 가져옵니다. 이 방법은 권장되지 않습니다. 대신 IndexReader를 사용하는 것이 좋습니다.

7.8.4. 분할 인덱스

경우에 따라 지정된 엔터티의 색인화된 데이터를 여러 Lucene 인덱스로 분할(하드)하는 것이 유용할 수 있습니다.

주의

분할이 단점을 능가하는 경우에만 구현해야 합니다. 단일 검색을 위해 모든 shard를 열어야 하므로 분할된 인덱스 검색은 일반적으로 더 느립니다.

분할 사용 사례는 다음과 같습니다.

- 단일 인덱스가 너무 커서 인덱스 업데이트 시간이 애플리케이션 다운 속도를 높입니다.
- 일반적인 검색은 고객, 지역 또는 애플리케이션에서 데이터를 자연스럽게 구분하는 경우와 같이 인덱스의 하위 집합에만 도달합니다.

기본적으로 분할 수는 구성되지 않는 한 활성화되지 않습니다. 이렇게 하려면 **hibernate.search.<indexName>.sharding_strategy.nbr_of_shards** 속성을 사용합니다.

예제: 인덱스 공유 활성화

이 예에서는 **5**개의 **shard**가 활성화됩니다.

```
hibernate.search.<indexName>.sharding_strategy.nbr_of_shards = 5
```

데이터를 하위 인덱스로 분할하는 작업은 **IndexShardingStrategy**입니다. 기본 분할 전략은 **ID** 문자열 표현의 해시 값(**FieldBridge**에서 생성)에 따라 데이터를 분할합니다. 이렇게 하면 상당히 분산된 분할이 가능합니다. 사용자 지정 **IndexShardingStrategy**를 구현하여 기본 전략을 교체할 수 있습니다. 사용자 정의 전략을 사용하려면 **hibernate.search.<indexName>.sharding_strategy** 속성을 설정해야 합니다.

예제: 사용자 정의 공유 전략 지정

```
hibernate.search.<indexName>.sharding_strategy = my.shardingstrategy.Implementation
```

IndexShardingStrategy 속성을 사용하면 쿼리를 실행할 **shard**를 선택하여 검색을 최적화할 수도 있습니다. 필터를 활성화하면 분할 전략이 쿼리(**IndexShardingStrategy.getIndexManagersForQuery**)에 응답하는 데 사용되는 **shard**의 하위 집합을 선택하여 쿼리 실행 속도를 높일 수 있습니다.

각 **shard**에는 독립적인 **IndexManager**가 있으므로 다른 디렉터리 공급자 및 백엔드 구성을 사용하도록 구성할 수 있습니다. 아래 예제에 있는 **indexManager** 엔터티의 **indexManager** 인덱스 이름은 **authenticate .0 ~three .4** 입니다. 즉, 각 **shard**에는 소유하는 인덱스의 이름과 그 뒤에 **. (dot)** 및 인덱스 번호가 있습니다.

예제: 엔터티 태그의 분할 구성

```
hibernate.search.default.indexBase = /usr/lucene/indexes
hibernate.search.Animal.sharding_strategy.nbr_of_shards = 5
hibernate.search.Animal.directory_provider = filesystem
hibernate.search.Animal.0.indexName = Animal00
hibernate.search.Animal.3.indexBase = /usr/lucene/sharded
hibernate.search.Animal.3.indexName = Animal03
```

위의 예에서 구성은 기본 **id** 문자열 해시 전략을 사용하고, **Base** 인덱스를 **5** 하위 색인으로 분할합니다. 모든 하위 색인은 파일 시스템 인스턴스이며 각 하위 색인이 저장되는 디렉토리는 다음과 같습니다.

- **sub-index 0**의 경우**: /usr/lucene/indexes/Animal00 (**공유 **indexBase**이지만 재정의된 **indexName)**
- **sub-index 1: /usr/lucene/indexes/Animal.1 (**공유 **indexBase,** 기본 **indexName)**
- **sub-index 2: /usr/lucene/indexes/Animal.2 (**공유 **indexBase,** 기본 **indexName)**
- **sub-index 3: /usr/lucene/shared/Animal03 (**재정 의된 **indexBase,** 재정의된 **indexName)**
- **sub-index 4: /usr/lucene/indexes/Animal.4 (**공유 **indexBase,** 기본 **indexName)**

IndexShardingStrategy를 구현할 때 필드를 사용하여 분할 선택 사항을 확인할 수 있습니다. 삭제, 삭제 및 삭제 작업을 처리하려면 구현에서 모든 필드 값이나 기본 식별자를 읽지 않고도 하나 이상의 인덱스를 반환해야 할 수 있습니다. 이 경우 단일 인덱스를 선택하는 것만으로는 충분하지 않으며 삭제 작업이 삭제될 문서가 포함된 모든 인덱스로 전파되도록 모든 인덱스가 반환되어야 합니다.

7.8.5. Lucene의 평가 사용자 정의

Lucene은 사용자가 **org.apache.lucene.search.Similarity**를 확장하여 점수 공식을 사용자 지정할 수 있습니다. 이 클래스에 정의된 추상화 방법은 문서 **d**에 대한 쿼리 **q** 점수를 계산하는 다음 공식의 요인과 일치합니다.

org.apache.lucene.search.Similarity를 확장하여 **Lucene**의 성과 공식을 사용자 지정합니다. 추상 방법은 다음과 같이 문서 **d** 에 대한 쿼리 **q** 의 점수를 계산하는 데 사용되는 공식과 일치합니다.

```
*score(q,d) = coord(q,d) · queryNorm(q) · ∑ ~t in q~ ( tf(t in d) ·
idf(t) ^2^ · t.getBoost() · norm(t,d) )*
```

요소	설명
tf(t ind)	문서(d)의 용어 (t)의 빈도 요인.
idf(t)	역순 문서 용어의 빈도.
coord(q,d)	지정된 문서에 있는 쿼리 용어 수에 따른 점수 인수입니다.
queryNorm(q)	쿼리 간에 점수를 만드는 데 사용되는 정규화 요소입니다.
t.getBoost()	필드 확대.
norm(t,d)	몇 가지 확장 시간(인덱싱 시간) 및 길이 요소를 캡슐화합니다.

이 설명서의 범위를 벗어난 이 설명서에서는 이 공식을 자세히 설명합니다. 자세한 내용은 Similarity의 Javadocs를 참조하십시오.

Hibernate Search는 Lucene의 유사성 계산을 수정하는 세 가지 방법을 제공합니다.

먼저 hibernate.search.similarity 속성을 사용하여 Similarity 구현의 완전히 지정된 클래스 이름을 지정하여 기본 유사성을 설정할 수 있습니다. 기본값은 org.apache.lucene.search.DefaultSimilarity입니다.

similarity 속성을 설정하여 특정 인덱스에 사용되는 유사성을 재정의할 수도 있습니다.

```
hibernate.search.default.similarity = my.custom.Similarity
```

마지막으로 @Similarity 주석을 사용하여 클래스 수준에서 기본 유사성을 재정의할 수 있습니다.

```
@Entity
@Indexed
@Similarity(impl = DummySimilarity.class)
public class Book {
...
}
```

예를 들어, 문서에 용어가 얼마나 자주 나타나는지는 중요하지 않다고 가정하겠습니다. 용어가 한 번 있는 문서는 여러 번 발생하는 문서와 동일해야 합니다. 이 경우 메서드 tf(float freq)의 사용자 지정 구현은 1.0을 반환해야 합니다.

주의

두 엔터티가 동일한 인덱스를 공유하는 경우 동일한 **Similarity** 구현을 선언해야 합니다. 동일한 클래스 계층 구조의 클래스는 항상 인덱스를 공유하므로 하위 유형에서 **Similarity** 구현을 재정의할 수 없습니다.

마찬가지로 인덱스 설정 및 클래스 수준 설정을 통해 충돌하는 것처럼 유사성을 정의하는 것은 바람직하지 않습니다. 이러한 구성은 거부됩니다.

7.8.6. 예외 처리 설정

Hibernate Search를 사용하면 인덱싱 프로세스 중에 예외를 처리하는 방법을 구성할 수 있습니다. 구성이 제공되지 않으면 기본적으로 로그 출력에 예외가 기록됩니다. 다음과 같이 예외 로깅 메커니즘을 명시적으로 선언할 수 있습니다.

```
hibernate.search.error_handler = log
```

동기 및 비동기 인덱싱에 대해 기본 예외 처리가 수행됩니다. **Hibernate Search**는 기본 오류 처리 구현을 쉽게 재정의할 수 있는 메커니즘을 제공합니다.

자체 구현을 제공하려면 **handle(ErrorContext** 컨텍스트) 메서드를 제공하는 **ErrorHandler** 인터페이스를 구현해야 합니다. **ErrorContext** 는 기본 예외로 인해 처리할 수 없는 기본 **LuceneWork** 인스턴스, 기본 예외 및 후속 **LuceneWork** 인스턴스에 대한 참조를 제공합니다.

```
public interface ErrorContext  {
  List<LuceneWork> getFailingOperations();
  LuceneWork getOperationAtFault();
  Throwable getThrowable();
  boolean hasErrors();
}
```

Hibernate Search를 사용하여 이 오류 핸들러를 등록하려면 구성 속성에서 **ErrorHandler** 구현의 정규화된 클래스 이름을 선언해야 합니다.

```
hibernate.search.error_handler = CustomerErrorHandler
```

7.8.7. Hibernate 검색 비활성화

필요에 따라 **Hibernate Search**를 부분적으로 또는 완전히 비활성화할 수 있습니다. 예를 들어 인덱스가 읽기 전용인 경우 **Hibernate Search**의 인덱싱을 비활성화하거나 자동으로 대신 인덱싱을 수동으로 수행하려고 할 수 있습니다. 또한 **Hibernate Search**를 완전히 비활성화하여 인덱싱 및 검색을 방지할 수 있습니다.

인덱싱 비활성화

Hibernate Search 인덱싱을 비활성화하려면 **indexing _strategy** 구성 옵션을 수동으로 변경한 다음 **JBoss EAP**를 다시 시작합니다.

```
hibernate.search.indexing_strategy = manual
```

Hibernate 검색 완전히 비활성화

Hibernate Search를 완전히 비활성화하려면 **autoregister_listeners** 구성 옵션을 **false** 로 변경한 다음 **JBoss EAP**를 다시 시작하여 모든 리스너를 비활성화합니다.

```
hibernate.search.autoregister_listeners = false
```

7.9. 모니터링

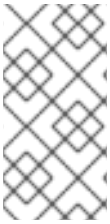
Hibernate Search는 **SearchFactory.getStatistics()** 를 통해 **Statistics** 객체에 대한 액세스를 제공합니다. 예를 들어 어떤 클래스가 인덱싱되고 인덱스에 있는 엔터티 수를 확인할 수 있습니다. 이 정보는 항상 사용할 수 있습니다. 그러나 구성에서 **hibernate.search.generate_statistics** 속성을 지정하면 총 및 평균 **Lucene** 쿼리 및 개체 로드 타이밍을 수집할 수도 있습니다.

자카르타 관리를 통해 통계 액세스

Jakarta Management를 통해 통계에 액세스할 수 있도록 하려면 **hibernate.search.jmx_enabled** 속성을 **true**로 설정합니다. 이렇게 하면 **StatisticsInfoMBean** 빈을 자동으로 등록하여 **Statistics** 오브젝트를 사용하여 통계에 액세스할 수 있습니다. 구성에 따라 **IndexingProgressMonitorMBean** 빈도 등록할 수 있습니다.

모니터링 인덱싱

대용량 인덱서 **API**를 사용하는 경우 **IndexingProgressMonitorMBean** 빈을 사용하여 인덱싱 진행 상황을 모니터링할 수 있습니다. 인덱싱이 진행되는 동안 빈은 **Jakarta Management**에만 바인딩됩니다.



참고

Jakarta Management Bean은 시스템 속성 **com.sun.management.jmxremote** 를 **true** 로 설정하여 **JConsole**을 사용하여 원격으로 액세스할 수 있습니다.

부록 A. 참고 자료

A.1. HIBERNATE 속성

표 A.1. persistence.xml 파일에서 구성 가능한 연결 속성

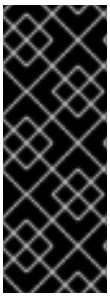
속성 이름	현재의	설명
javax.persistence.jdbc.driver	**org.hsqldb.jdbcDriver**	사용할 JDBC 드라이버의 클래스 이름입니다.
javax.persistence.jdbc.user	SA	사용자 이름.
javax.persistence.jdbc.password		암호.
javax.persistence.jdbc.url	**jdbc:hsqldb:.**	JDBC 연결 URL입니다.

표 A.2. Hibernate 구성 속성

속성 이름	설명
hibernate.dialect	Hibernate **org.hibernate.dialect.Dialect** 의 클래스 이름입니다. Hibernate에서 특정 관계형 데이터베이스에 최적화된 SQL을 생성할 수 있도록 합니다. 대부분의 경우 Hibernate는 JDBC 드라이버에서 반환한 JDBC 메타데이터를 기반으로 올바른 **org.hibernate.dialect.Dialect** 구현을 선택할 수 있습니다.
hibernate.show_sql	부울. 모든 SQL 문을 콘솔에 씁니다. 이는 로그 범주를 **debug** 하도록 **org.hibernate.SQL** 을 설정하는 대신 사용됩니다.
hibernate.format_sql	부울. 로그와 콘솔에서 SQL을 인쇄합니다.
hibernate.default_schema	생성된 SQL에서 지정된 스키마/표 공간을 사용하여 정규화되지 않은 테이블 이름을 지정합니다.
hibernate.default_catalog	생성된 SQL에서 지정된 카탈로그로 정규화되지 않은 테이블 이름을 정규화합니다.
hibernate.session_factory_name	org.hibernate.SessionFactory는 Java Naming 및 Directory Interface에서 이 이름에 자동으로 바인딩됩니다. 예를 들면 **jndi/composite/name**.Jpa입니다.
hibernate.max_fetch_depth	단일 종료 연관(일대일, 다대일)의 외부 조인 가져오기 트리의 최대 깊이를 설정합니다. **0** 은 기본 외부 조인 가져오기를 비활성화합니다. 권장 값은 **0** 에서 **3** 사이입니다.

속성 이름	설명
hibernate.default_batch_fetch_size	연관을 가져오는 Hibernate 배치의 기본 크기를 설정합니다. 권장 값은 **4**,**8**, **16** 입니다.
hibernate.default_entity_mode	이 **SessionFactory** 에서 연 모든 세션에 대한 엔터티 표현의 기본 모드를 설정합니다. 값이 다음과 같습니다. **dynamic-map**,**dom4j**,**pojo**.
hibernate.order_updates	부울. 업데이트되는 항목의 기본 키 값에 따라 Hibernate가 SQL 업데이트를 주문하도록 합니다. 그러면 고도의 동시 시스템에서 트랜잭션 교착 상태가 줄어듭니다.
hibernate.generate_statistics	부울. 활성화된 경우 Hibernate는 성능 튜닝에 유용한 통계를 수집합니다.
hibernate.use_identifier_rollback	부울. 활성화된 경우 개체가 삭제될 때 생성된 식별자 속성이 기본값으로 재설정됩니다.
hibernate.use_sql_comments	부울. 켜진 경우 Hibernate는 더 쉬운 디버깅을 위해 SQL 내부에 주석을 생성합니다. 기본값은 **false** 입니다.
hibernate.id.new_generator_mappings	부울. 이 속성은 @GeneratedValue를 사용할 때 관련이 있습니다. 새로운 IdentifierGenerator 구현이 javax.persistence.GenerationType.AUTO, javax.persistence.GenerationType.TABLE 및 javax.persistence.GenerationType.SEQUENCE에 사용되는지 여부를 나타냅니다. 기본값은 **true** 입니다.
hibernate.ejb.naming_strategy	Hibernate EntityManager를 사용할 때 org.hibernate.cfg.NamingStrategy 구현을 선택합니다. **hibernate.ejb.naming_strategy** 는 더 이상 Hibernate 5.0에서 지원되지 않습니다. 사용된 경우 더 이상 지원되지 않으며 분할 ImplicitNamingStrategy 및 PhysicalNamingStrategy가 제거되었음을 나타내는 사용 중단 메시지가 기록됩니다. 애플리케이션이 EntityManager를 사용하지 않는 경우 다음 지침에 따라 NamingStrategy를 구성합니다. Hibernate 참조 설명서 - 전략 명명. MetadataBuilder를 사용한 기본 부트 스트랩 및 암시적 명명 전략을 적용하는 예제는 Hibernate 5.0 문서의 http://docs.jboss.org/hibernate/orm/5.0/userguide/html_single/Hibernate_User_Guide.html#bootstrap-native-metadata 을 참조하십시오. 물리적 명명 전략은 **MetadataBuilder.applyPhysicalNamingStrategy()** 를 사용하여 적용할 수 있습니다. **org.hibernate.boot.MetadataBuilder** 에 대한 자세한 내용은 https://docs.jboss.org/hibernate/orm/5.0/javadocs/ 을 참조하십시오.

속성 이름	설명
hibernate.implicit_naming_strategy	사용할 **org.hibernate.boot.model.naming.ImplicitNamingStrategy** 클래스를 지정합니다. **hibernate.implicit_naming_strategy** 를 사용하여 ImplicitNamingStrategy를 구현하는 사용자 지정 클래스를 구성할 수도 있습니다. 다음 단축 이름은 이 설정에 대해 정의됩니다. • **default** - **ImplicitNamingStrategyJpaCompliantImpl** • **jpa** - **ImplicitNamingStrategyJpaCompliantImpl** • **legacy-jpa** - **ImplicitNamingStrategyLegacyJpaImpl** • **legacy-hbm** - **ImplicitNamingStrategyLegacyHbmImpl** • **component-path** - **ImplicitNamingStrategyComponentPathImpl** 기본 설정은 **기본** 짧은 이름의 **ImplicitNamingStrategy** 에 의해 정의됩니다. 기본 설정이 비어 있는 경우 대체는 **ImplicitNamingStrategyJpaCompliantImpl** 을 사용하는 것입니다.
hibernate.physical_naming_strategy	데이터베이스 개체 이름에 대한 물리적 이름 지정 규칙을 적용하기 위한 플러그 방식 전략 계약입니다. 사용할 PhysicalNamingStrategy 클래스를 지정합니다. **PhysicalNamingStrategyStandardImpl** 은 기본적으로 사용됩니다. **hibernate.physical_naming_strategy** 를 사용하여 PhysicalNamingStrategy를 구현하는 사용자 지정 클래스를 구성할 수도 있습니다.



중요

hibernate.id.new_generator_mappings의 경우 새 애플리케이션은 기본값인 true 를 유지해야 합니다. Hibernate 3.3.x를 사용한 기존 애플리케이션은 시퀀스 개체 또는 테이블 기반 생성기를 계속 사용하고 이전 버전과의 호환성을 유지하도록 false로 변경해야 할 수 있습니다.

표 A.3. Hibernate JDBC 및 연결 속성

속성 이름	설명
hibernate.jdbc.fetch_size	JDBC 가져오기 크기를 결정하는 0이 아닌 값입니다 (calls **statement.setFetchSize()**).

속성 이름	설명
hibernate.jdbc.batch_size	0이 아닌 값을 사용하면 Hibernate에서 JDBC2 배치 업데이트를 사용할 수 있습니다. 권장 값은 **5** 에서 **30** 사이입니다.
hibernate.jdbc.batch_versioned_data	부울. JDBC 드라이버가 **executeBatch()** 에서 올바른 행 수를 반환하는 경우 이 속성을 **true** 로 설정합니다. 그런 다음 Hibernate는 자동으로 버전화된 데이터에 배치된 DML을 사용합니다. 기본값은 **false** 입니다.
hibernate.jdbc.factory_class	사용자 지정 org.hibernate.jdbc.Batcher를 선택합니다. 대부분의 애플리케이션에는 이 구성 속성이 필요하지 않습니다.
hibernate.jdbc.use_scrollable_resultset	부울. Hibernate에서 JDBC2 스크롤 가능한 결과 사용 가능. 이 속성은 사용자가 제공하는 JDBC 연결을 사용하는 경우에만 필요합니다. Hibernate는 연결 메타데이터를 사용하지 않습니다.
hibernate.jdbc.use_streams_for_binary	부울. 이는 시스템 수준 속성입니다. **바이너리** 또는 **직렬화 가능한** 유형을 JDBC에/에서 작성할 때 스트림을 사용합니다.
hibernate.jdbc.use_get_generated_keys	부울. **PreparedStatement.getGeneratedKeys()** 를 사용하여 삽입 후 기본적으로 생성된 키를 검색합니다. JDBC3+ 드라이버와 JRE1.4 이상이 필요합니다. JDBC 드라이버에 Hibernate 식별자 생성기에 문제가 있는 경우 false로 설정합니다. 기본적으로 연결 메타데이터를 사용하여 드라이버 기능을 확인합니다.
hibernate.connection.provider_class	Hibernate에 JDBC 연결을 제공하는 사용자 지정 org.hibernate.connection.ConnectionProvider의 클래스 이름입니다.
hibernate.connection.isolation	JDBC 트랜잭션 격리 수준을 설정합니다. java.sql.Connection에 의미 있는 값이 있는지 확인하지만 대부분의 데이터베이스는 모든 격리 수준을 지원하지 않으며 일부는 표준이 아닌 추가 격리를 정의합니다. 표준 값은 **1, 2, 4, 8** 입니다.
hibernate.connection.autocommit	부울. 이 속성은 사용하지 않는 것이 좋습니다. JDBC 풀링 연결에 대해 자동 커밋을 활성화합니다.

속성 이름	설명
hibernate.connection.release_mode	Hibernate에서 JDBC 커넥션을 릴리스해야 하는 시기를 지정합니다. 기본적으로 세션이 명시적으로 종료되거나 연결이 끊어질 때까지 JDBC 연결이 유지됩니다. 기본 값은 **Jakarta** 트랜잭션 및 CMT 트랜잭션 전략에 **after_statement** 를 선택하고 JDBC 트랜잭션 전략에 **after_transaction** 을 선택합니다. 사용 가능한 값은 auto(기본값), on_close, **after_transaction**,**after_statement** 입니다. 이 설정은 **SessionFactory.openSession** 에서 반환된 세션에만 영향을 미칩니다. **SessionFactory.getCurriteSession** 을 통해 얻은 세션의 경우, 사용을 위해 구성된 **CurrentSessionContext** 구현은 해당 세션의 연결 릴리스 모드를 제어합니다.
hibernate.connection.*<propertyName>*	JDBC 속성 *<propertyName>* 을 **DriverManager.getConnection()** 에 전달합니다.
hibernate.jndi.*<propertyName>*	*<propertyName>* 속성을 Java Naming 및 Directory Interface **InitialContextFactory** 로 전달합니다.

표 A.4. Hibernate 캐시 속성

속성 이름	설명
hibernate.cache.region.factory_class	사용자 지정 **CacheProvider** 의 클래스 이름입니다.
hibernate.cache.use_minimal_puts	부울. 두 번째 수준 캐시 작업을 최적화하여 더 빈번한 읽기로 쓰기를 최소화합니다. 이 설정은 클러스터된 캐시에 가장 유용하며 Hibernate3에서는 클러스터된 캐시 구현에 기본적으로 활성화됩니다.
hibernate.cache.use_query_cache	부울. 쿼리 캐시를 활성화합니다. 개별 쿼리는 계속 캐시 가능해야 합니다.
hibernate.cache.use_second_level_cache	부울. **<cache>** 매핑을 지정하는 클래스에 대해 기본적으로 활성화되어 있는 두 번째 수준 캐시를 완전히 비활성화하는 데 사용됩니다.
hibernate.cache.query_cache_factory	사용자 정의 **QueryCache** 인터페이스의 클래스 이름입니다. 기본값은 기본 제공 **StandardQueryCache** 입니다.
hibernate.cache.region_prefix	두 번째 수준 캐시 지역 이름에 사용할 접두사입니다.

속성 이름	설명
hibernate.cache.use_structured_entries	부울. Hibernate가 사용자에게 친숙한 형식으로 두 번째 수준 캐시에 데이터를 저장하도록 합니다.
hibernate.cache.default_cache_concurrency_strategy	@Cacheable 또는 @Cache가 사용되는 경우 사용할 기본 org.hibernate.annotations.CacheConcurrencyStrategy의 이름을 지정하는 데 사용되는 설정입니다. **@Cache(strategy="..")** 는 이 기본값을 재정의하는 데 사용됩니다.

표 A.5. Hibernate 트랜잭션 속성

속성 이름	설명
hibernate.transaction.factory_class	Hibernate 트랜잭션 **API** 와 함께 사용할 **TransactionFactory** 의 클래스 이름입니다. 기본값은 **JDBCTransactionFactory**입니다.
jta.UserTransaction	애플리케이션 서버에서 자카르타 트랜잭션 **사용자 전송을 얻기 위해 JTATransactionFactory 에서** 사용하는 Java 네이밍 및 디렉터리 인터페이스 이름입니다.
hibernate.transaction.manager_lookup_class	**TransactionManagerLookup의 클래스 이름입니다**. JVM 수준 캐싱이 활성화되거나 자카르타 트랜잭션 환경에서 hilo 생성기를 사용할 때 필요합니다.
hibernate.transaction.flush_before_completion	부울. 활성화된 경우 트랜잭션을 완료하기 전 단계에서 세션이 자동으로 플러시됩니다. 기본 제공 및 자동 세션 컨텍스트 관리가 선호됩니다.
hibernate.transaction.auto_close_session	부울. 활성화된 경우 트랜잭션 완료 단계 동안 세션이 자동으로 종료됩니다. 기본 제공 및 자동 세션 컨텍스트 관리가 선호됩니다.

표 A.6. 기타 Hibernate 속성

속성 이름	설명
hibernate.current_session_context_class	"current" **세션** 의 범위를 위한 사용자 지정 전략을 제공합니다. 값에는 **jta**,**thread**,**managed**,**custom.Class** 가 있습니다.

속성 이름	설명
hibernate.query.factory_class	HQL 구문 분석기 구현을 선택합니다 **. org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory** 또는 **org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory**.
hibernate.query.substitutions	Hibernate 쿼리의 토큰에서 SQL 토큰으로 매핑하는 데 사용됩니다(토큰은 기능 또는 리터럴 이름일 수 있음). 예를 들면 **hqlLiteral=SQL_LITERAL, hqlœction=SQLFUNC** 입니다.
hibernate.query.conventional_java_constants	Java 상수가 Java 명명 규칙을 따르는지 여부를 나타냅니다. 기본값은 **false** 입니다. 기존 애플리케이션은 기존 Java 상수가 애플리케이션에서 사용되는 경우에만 **true** 로 설정할 수 있습니다. 이 값을 **true** 로 설정하면 성능이 크게 향상됩니다. 그러면 Hibernate에서 별칭이 Java 명명 규칙을 따르는지 확인함으로써 Java 상수로 취급되는지 여부를 확인할 수 있기 때문입니다. 이 속성이 **false** 로 설정되면 Hibernate는 애플리케이션의 오버헤드인 클래스로 별칭을 로드하여 Java 상수로 취급합니다. 별칭이 클래스로 로드되지 않으면 Hibernate는 별칭을 Java 상수로 처리합니다.
hibernate.hbm2ddl.auto	**SessionFactory** 가 생성될 때 스키마 DDL을 데이터베이스로 자동 확인하거나 내보냅니다. **create-drop** 을 사용하면 **SessionFactory** 가 명시적으로 닫힐 때 데이터베이스 스키마가 삭제됩니다. 속성 값 옵션은 **validate**,**update**,**create, create -drop**입니다.
hibernate.hbm2ddl.import_files	SessionFactory 생성 중에 실행된 SQL DML 문이 포함된 선택적 파일의 쉼표로 구분된 이름입니다. 이는 테스트 또는 시연에 유용합니다. 예를 들어 INSERT 문을 추가하여 데이터베이스를 배포할 때 최소한의 데이터 집합으로 채울 수 있습니다. 예제 값은 **/ruless.sql,/dogs.sql** 입니다. 지정된 파일의 설명이 다음 파일의 설명보다 먼저 실행되므로 파일 순서가 중요합니다. 이러한 설명은 스키마가 생성되는 경우에만 실행됩니다(예: **hibernate.hbm2ddl.auto** 가 **create 또는 create -drop** 로 설정된 경우).

속성 이름	설명
hibernate.hbm2ddl.import_files_sql_extractor	사용자 지정 ImportSqlCommandExtractor의 클래스 이름입니다. 기본값은 내장된 SingleLineSqlCommandExtractor입니다. 이는 각 가져오기 파일에서 단일 SQL 문을 추출하는 전용 구문 분석기를 구현하는 데 유용합니다. Hibernate는 또한 MultiplelinesSqlCommandExtractor를 제공하여 여러 줄에 분산된 명령/컴파일과 인용된 문자열을 지원합니다(각 문 끝에 필수 세미콜론).
hibernate.bytecode.use_reflection_optimizer	부울. 이것은 시스템 수준 속성으로, **hibernate.cfg.xml** 파일에서 설정할 수 없습니다. 런타임을 반영하는 대신 바이트코드 조작을 활성화합니다. 경우에 따라 문제 해결 시 유용할 수 있습니다. Hibernate에는 최적화 도구가 꺼져 있어도 항상 **cglib** 또는 **javassist** 가 필요합니다.
hibernate.bytecode.provider	**javassist** 또는 **cglib** 둘 다 바이트 조작 엔진으로 사용할 수 있습니다. 기본값은 **javassist** 입니다. 값은 **javassist** 또는 **cglib** 입니다.

표 A.7. Hibernate SQL Dialects (hibernate.dialect)

RDBMS	전화 번호
DB2	**org.hibernate.dialect.DB2Dialect**
DB2 AS/400	**org.hibernate.dialect.DB2400Dialect**
DB2 OS390	**org.hibernate.dialect.DB2390Dialect**
타바마	**org.hibernate.dialect.FirebirdDialect**
FrontBase	**org.hibernate.dialect.FrontbaseDialect**
H2 데이터베이스	**org.hibernate.dialect.H2Dialect**
HypersonicSQL	**org.hibernate.dialect.HSQLDialect**
Informix	**org.hibernate.dialect.InformixDialect**
Ingres	**org.hibernate.dialect.IngresDialect**
Interbase	**org.hibernate.dialect.InterbaseDialect**
MariaDB 10	**org.hibernate.dialect.MariaDB10Dialect**

RDBMS	전화 번호
MariaDB Galera Cluster 10	**org.hibernate.dialect.MariaDB10Dialect**
Mckoi SQL	**org.hibernate.dialect.MckoiDialect**
Microsoft SQL Server 2000	**org.hibernate.dialect.SQLServerDialect**
Microsoft SQL Server 2005	**org.hibernate.dialect.SQLServer2005Dialect**
Microsoft SQL Server 2008	**org.hibernate.dialect.SQLServer2008Dialect**
Microsoft SQL Server 2012	**org.hibernate.dialect.SQLServer2012Dialect**
Microsoft SQL Server 2014	**org.hibernate.dialect.SQLServer2012Dialect**
Microsoft SQL Server 2016	**org.hibernate.dialect.SQLServer2012Dialect**
MySQL5	**org.hibernate.dialect.MySQL5Dialect**
MySQL5.5	**org.hibernate.dialect.MySQL55Dialect**
MySQL5.7	**org.hibernate.dialect.MySQL57Dialect**
Oracle (모든 버전)	**org.hibernate.dialect.OracleDialect**
Oracle 9i	**org.hibernate.dialect.Oracle9iDialect**
Oracle 10g	**org.hibernate.dialect.Oracle10gDialect**
Oracle 11g	**org.hibernate.dialect.Oracle10gDialect**
Oracle 12c	**org.hibernate.dialect.Oracle12cDialect**
Pointbase	**org.hibernate.dialect.PointbaseDialect**
PostgreSQL	**org.hibernate.dialect.PostgreSQLDialect**
PostgreSQL 9.2	**org.hibernate.dialect.PostgreSQL9Dialect**
PostgreSQL 9.3	**org.hibernate.dialect.PostgreSQL9Dialect**
PostgreSQL 9.4	**org.hibernate.dialect.PostgreSQL94Dialect**
Postgres Plus 고급 서버	**org.hibernate.dialect.PostgresPlusDialect**
진행 상황	**org.hibernate.dialect.ProgressDialect**

RDBMS	전화 번호
SAP DB	**org.hibernate.dialect.SAPDBDialect**
Sybase	**org.hibernate.dialect.SybaseASE15Dialect**
Sybase 15.7	**org.hibernate.dialect.SybaseASE157Dialect**
Sybase 16	**org.hibernate.dialect.SybaseASE157Dialect**
Sybase Anywhere	**org.hibernate.dialect.SybaseAnywhereDialect**

중요

애플리케이션 데이터베이스의 **hibernate.dialect** 속성은 올바른 **org.hibernate.dialect.Dialect** 하위 클래스로 설정해야 합니다. 전화선이 지정되면 **Hibernate**는 다른 일부 속성에 대해 적절한 기본값을 사용합니다. 즉 수동으로 지정할 필요가 없습니다.

2024-02-09에 최종 업데이트된 문서