Introdução ao Hibernate Search

[Facebook](https://www.facebook.com/sharer/sharer.php?u=http://www.devmedia.com.br/introducao-ao-hibernate-search/29739" \o "Compartilhe no Facebook" \t "_blank) [Twitter](https://twitter.com/home?status=Hibernate%20Search%0D%0Ahttp://www.devmedia.com.br/introducao-ao-hibernate-search/29739" \o "Compartilhe no Twitter" \t "_blank)

 (0)  (1)

Veja neste artigo o que é o Hibernate Search e como podemos criar uma aplicação com este framework.

Hibernate Search permite aos desenvolvedores utilizarem facilmente o Apache Lucene com nosso modelo de objetos do Hibernate. Lucene é uma engine de pesquisa de código aberto livremente disponível. O Apache Lucene analisa uma grande quantidade de textos e criar uma pesquisa indexada. O Lucene, por si só, é um componente que provê funcionalidades de pesquisa no seu núcleo principal, incluindo análise, indexação e consulta. Nossa aplicação se integra com o Lucene para fornecer conteúdo para indexação e consultas para executar contra a indexação.

Isso pode ser frustrante quando se trabalha com Hibernate porque há uma incompatibilidade entre Lucene e objetos Java que usam Hibernate, similar a incompatibilidade objeto/relacional. No entanto, o Hibernate Search vem para terminar com essa inconsistência.

Por trás da funcionalidade de pesquisa do Hibernate Search está o Apache Lucene, uma biblioteca de código-fonte aberto para indexação e pesquisa. O Apache Lucene é considerado um projeto Java com uma rica história de inovação, embora ainda possa ser portada para outras linguagens de programação, não se limitando apenas ao Java. O Apache Lucene é adotado em uma grande variedade de projetos na indústria, entre eles se destacam a Disney e o Twitter.

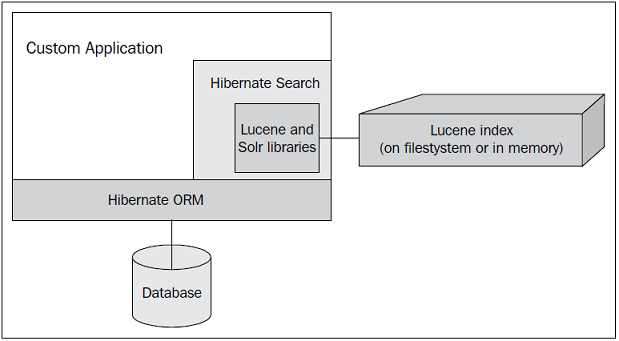
Um projeto inter-relacionado ao Apache Lucene é o Apache Solr, que é um servidor de pesquisa standalone baseado no Lucene.

No restante do artigo veremos mais sobre o Hibernate Search, seu funcionamento, as suas principais características e exemplos.

**Funcionamento**

O Hibernate Search é um “empacotador magro” ou um thin wrapper que abrange o Lucene e um componente opcional Solr. Ele estende o núcleo principal do Hibernate ORM, o mais utilizado framework de mapeamento objeto/relacional para Persistência Java.

A **Figura 1**mostra o relacionamento entre todos esses componentes citados:



**Figura 1**. Relacionamento entre os componentes.

As últimas versões do Hibernate Search envolvem dois papeis principais:

* Primeiro, traduzir objetos de informação do Hibernate em informações que o Lucene possa usar para construir pesquisas indexadas.
* Traduzir os resultados para pesquisa do Lucene em um formato familiar para o Hibernate.

O Hibernate Search esconde a maior parte do baixo nível realizado através do Lucene.

**Aplicação de Exemplo**

Para incorporar o Hibernate Search na nossa aplicação precisamos realizar três passos:

1. Adicionar informação às nossas classes de entidades para que o Lucene saiba como indexa-las.
2. Escrever uma ou mais consultas nas porções relevantes da aplicação.
3. Configurar o projeto para que as dependências necessárias e a configuração para o Hibernate Search estejam disponíveis primeiramente.

Portanto, a primeira situação a ser feita é criar uma classe de entidade como mostrada abaixo. A classe de entidade exemplifica um login que exige um nome de usuário e senha. Observe a **Listagem 1**.

**Listagem 1**. Exemplo de uma classe de entidade.

package exemplo.hibernatesearch;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.Table;

import org.hibernate.annotations.GenericGenerator;

@Entity

@Table(name="login", schema="public")

public class Login {

@Id

@GeneratedValue(generator="increment")

@GenericGenerator(name="increment", strategy = "increment")

private int id;

@Column

private String usuario;

@Column

private String senha;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getUsuario() {

return usuario;

}

public void setUsuario(String usuario) {

this.usuario = usuario;

}

public String getSenha() {

return senha;

}

public void setSenha(String senha) {

this.senha = senha;

}

}

A classe acima diz ao Hibernate para que a classe seja mapeada para uma tabela de uma base de dados.

Agora devemos preparar essa classe de entidade para o Hibernate Search. Dessa forma, o Hibernate Search saberá como gerenciar essa classe com o Lucene.

Para isto, devemos adicionar a anotação @Indexed na classe, conforme mostrado na **Listagem 2**.

**Listagem 2**. Adicionando a anotação para o Hibernate Search.

@Entity

@Indexed

@Table(name="login", schema="public")

public class Login {

…

}

Este exemplo declara que o Lucene deveria construir e usar uma indexação para este classe de entidade. Esta anotação é opcional. Quando escrevermos uma aplicação de larga escala, muitas das classes de entidades podem não ser relevantes para pesquisa. O Hibernate Search apenas necessita dizer ao Lucene sobre esses tipos que serão considerados pesquisáveis.

Após isso, devemos declarar pontos de informações que serão pesquisáveis através da anotação @Field, conforme mostra o exemplo da **Listagem 3**.

**Listagem3**.Adicionando a anotação @Field à nossa classe

package exemplo.hibernatesearch;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.Table;

import org.hibernate.search.annotations.Field;

import org.hibernate.annotations.GenericGenerator;

@Entity

@Table(name="login", schema="public")

public class Login {

@Id

@GeneratedValue(generator="increment")

@GenericGenerator(name="increment", strategy = "increment")

private int id;

@Column

@Field

private String usuario;

@Column

private String senha;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getUsuario() {

return usuario;

}

public void setUsuario(String usuario) {

this.usuario = usuario;

}

public String getSenha() {

return senha;

}

public void setSenha(String senha) {

this.senha = senha;

}

}

Podemos notar que apenas o nome de usuário obteve a anotação @Field. Isso ocorre porque não temos o interesse de pesquisar por id ou senha, por isso esses atributos não são anotados.

Para que possamos testar a nossa aplicação criaremos a classe e método que estão na **Listagem 4**.

**Listagem 4**. Construindo uma classe de teste inicial.

package exemplo.hibernatesearch;

import java.util.List;

import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

import org.hibernate.service.ServiceRegistry;

import org.hibernate.service.ServiceRegistryBuilder;

public class TesteHibernate {

//método thread-safe

private static synchronized Session openSession() {

Configuration conf = new Configuration();

conf.configure();

ServiceRegistry serviceRegistry = new

ServiceRegistryBuilder().applySettings

(conf.getProperties()).buildServiceRegistry();

SessionFactory sessionFactory =

conf.buildSessionFactory(serviceRegistry);

Session session = sessionFactory.openSession();

return session;

}

public void teste() {

}

}

O método teste conterá a lógica para sessões e manipulação da base de dados. Na **Listagem 5** inserimos um dado qualquer na base de dados.

**Listagem 5**. Inserindo dados na base de dados.

public void insere() {

Session session = openSession();

try {

session.beginTransaction();

Login p = new Login();

p.setSenha("user1");

p.setUsuario("senha1");

//salva usuario criado acima

session.save(p);

//comita a transacao

session.getTransaction().commit();

} catch ( HibernateException e ) {

if ( session.getTransaction() != null )

session.getTransaction().rollback();

} finally {

session.close();

}

}

Feito isso, podemos agora escrever nosso primeiro código de pesquisa utilizando uma simples página index.html, conforme a **Listagem 6**.

**Listagem 6**. Tela de pesquisa para nome de usuário.

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Página de Pesquisa</title>

</head>

<body>

<h1>Bem-vindo à Página de Exemplo</h1>

Entre com um nome de usuário para pesquisa:

<form action="pesquisa" method="post">

<div id="pesquisa">

<div>

<input type="text" name="nomeUsuario" />

<input type="submit" value="Pesquisar" />

</div>

</div>

</form>

</body>

</html>

Também podemos fazer uma tela swing com os dados acima chamando o nosso método teste() que será definido abaixo. Não se esqueça de chamar uma vez o método insere() para adicionar alguma informação na base de dados para fins de teste.

Agora podemos executar a nossa consulta para a pesquisa. O código da **Listagem 7** demonstra como poderíamos fazer isso.

**Listagem 7**. Criando uma sessão do Hibernate Search

import org.hibernate.Session;

import org.hibernate.search.FullTextSession;

import org.hibernate.search.Search;

//mais códigos

Session session = openSession();

FullTextSessionfullTextSession = Search.getFullTextSession(session);

fullTextSession.beginTransaction();

Agora que possuímos uma sessão do Hibernate Serach podemos realizar a pesquisa com Lucene Search, conforme a **Listagem 8.**

**Listagem 8**. Criando uma pesquisa com Lucene Search.

import org.hibernate.search.query.dsl.QueryBuilder;

//Mais códigos

String nomeUsuario = request.getParameter("nomeUsuario");

QueryBuilder queryBuilder = fullTextSession.getSearchFactory()

.buildQueryBuilder().forEntity(Login.class).get();

org.apache.lucene.search.Query luceneQuery =

queryBuilder.keyword().onFields("usuario").matching

(nomeUsuario).createQuery();

//Mais Códigos

QueryBuilder é utilizado para construir consultas invocando uma classe de entidade em particular. No exemplo acima construímos uma consulta do tipo keyword no campo "nomeUsuario" combinando palavras-chave em nomeUsuario.

O objeto org.apache.lucene.search.Query é traduzido pelo Hibernate Search em uma busca do Lucene. Vale salientar que isso ocorre em ambas as direções. Resultados do Lucene pode ser traduzido em um objeto org.hibernate.Query e o mesmo pode ser utilizado como qualquer consulta normal de uma base de dados. Observe a **Listagem 9**.

**Listagem 9**. Realizando uma consulta no Hibernate.

org.hibernate.Query hibernateQuery =

fullTextSession.createFullTextQuery(luceneQuery, Login.class);

List<App> apps = hibernateQuery.list();

request.setAttribute("apps", apps);

No exemplo acima pesquisamos todas as entidades Login que foram encontradas na nossa consulta e colocamos num servlet request.

Por fim, poderíamos exibir os dados encontrados através da JSP presente na **Listagem 10**.

**Listagem 10**. Exibindo os resultados.

<%@ page language="java" contentType="text/html;

charset=UTF-8" pageEncoding="UTF-8"%>

<%@ tagliburi="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>

<head>

<title>Pesquisa Nome de Usuário</title>

</head>

<body>

<h1>Resultados da Pesquisa</h1>

<table>

<tr>

<td><b>Nome de Usuário:</b></td>

</tr>

<c:forEachvar="app" items="${apps}">

<tr>

<td>${app.nomeUsuario}</td>

</tr>

</c:forEach>

</table>

</body>

</html>

Com isso, neste artigo vimos o que é o Hibernate Search, como se dá o seu funcionamento interno, e quais são seus principais componentes. Por fim fizemos uma pequena aplicação demonstrando o seu uso.