



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Hibernate Search by Example

Explore the Hibernate search system and use its extraordinary search features in your own applications

Steve Perkins

[PACKT] open source*
PUBLISHING community experience distilled

Hibernate Search by Example

Explore the Hibernate Search system and use its extraordinary search features in your own applications

Steve Perkins



Hibernate Search by Example

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: March 2013

Production Reference: 1140313

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK..

ISBN 978-1-84951-920-5

www.packtpub.com

Cover Image by J. Blaminsky (milak6@wp.pl)

Credits

Author

Steve Perkins

Project Coordinator

Amigya Khurana

Reviewers

Shaozhuang Liu

Murat Yener

Proofreader

Ting Baker

Acquisition Editor

Joanne Fitzpatrick

Indexer

Monica Ajmera

Commissioning Editor

Meeta Rajani

Graphics

Sheetal Aute

Technical Editors

Amit Ramadas

Lubna Shaikh

Production Coordinator

Shantanu Zagade

Cover Work

Shantanu Zagade

About the Author

Steve Perkins is a Java developer based in Atlanta, GA, USA. Steve has been working with Java in the web and systems integration contexts for 15 years, for clients ranging from commerce and finance to media and entertainment. He has been using Hibernate intensively for over seven years, and is interested in best practices for data modeling and application design.

Apart from coding, Steve also has a keen interest in the subject of software patents, which eventually led to a law degree and becoming a licensed attorney. Steve co-authored *In the Aftermath of In re Bilski*, published in 2009, and *In the Aftermath of Bilski v. Kappos*, published in 2010, for the *Practicing Law Institute Handbook Series*.

Steve lives in Atlanta with his wife, Amanda, their son, Andrew, and more musical instruments than he has free time to play. You can visit his website at steveperkins.net and follow him on Twitter at [@stevedperkins](https://twitter.com/stevedperkins).

This book is dedicated to my wife, Amanda, for supporting me through the experience of a new baby and a new book all in the same year. We are very grateful for the support and encouragement of all our family and friends.

Thanks to the reviewers and the editorial staff at Packt Publishing. Last but not least, I deeply appreciate every hiring manager whoever took a chance on me. I would have nothing to write about today if it weren't for a handful of key people throwing me into the deep end and letting me swim.

About the Reviewers

Shaozhuang Liu has over seven years of experience in Java EE, and now as a senior member of the Hibernate development team, his main focus is the Hibernate ORM open source project. He's also interested in building cool things based on open source hardware, such as Arduino and Raspberry Pi. When he is not coding, traveling and snowboarding are the two favorite activities he enjoys.

Murat Yener completed his BS and MS degree at Istanbul Technical University. He has taken part in several projects still in use at the ITU Informatics Institute. He has worked for Isbank's Core Banking Exchange project as a J2EE developer. He has also designed and completed several projects still in the market by Muse Systems. He has worked for TAV Airports Information Technologies as an Enterprise Java and Flex developer. He has worked HSBC as the Project Leader responsible for Business Processes and Rich client user interfaces. He is currently employed at Eteration A.S. as Principal Mentor, working on several projects including Eclipse Libra Tools, GWT, and Mobile applications (both on Android and iOS).

He is also leading Google Technology User Group Istanbul since 2009, and is a regular speaker at conferences, such as JavaOne, EclipseCon, EclipsIst, and GDG meetings.

I would like to thank Naci Dai for being my mentor and providing the best work environment, Daniel Kurka for developing mgwt, the best mobile platform I have ever worked on, and Nilay Coskun for all her support.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Your First Application	7
Creating an entity class	8
Preparing the entity for Hibernate Search	10
Loading the test data	11
Writing the search query code	14
Selecting a build system	17
Setting up the project and importing Hibernate Search	19
Running the application	21
Summary	26
Chapter 2: Mapping Entity Classes	27
Choosing an API for Hibernate ORM	27
Field mapping options	30
Multiple mappings for the same field	31
Mapping numeric fields	31
Relationships between entities	32
Associated entities	32
Querying associated entities	35
Embedded objects	36
Partial indexing	39
The programmatic mapping API	40
Summary	42

Chapter 3: Performing Queries	43
Mapping API versus query API	43
Using JPA for queries	44
Setting up a project for Hibernate Search and JPA	45
The Hibernate Search DSL	46
Keyword query	47
Fuzzy search	48
Wildcard search	50
Exact phrase query	50
Range query	52
Boolean (combination) queries	53
Sorting	54
Pagination	56
Summary	57
Chapter 4: Advanced Mapping	59
Bridges	59
One-to-one custom conversion	60
Mapping date fields	60
Handling null values	60
Custom string conversion	61
More complex mappings with FieldBridge	64
Splitting a single variable into multiple fields	65
Combining multiple properties into a single field	66
TwoWayFieldBridge	67
Analysis	68
Character filtering	69
Tokenization	69
Token filtering	70
Defining and selecting analyzers	70
Static analyzer selection	71
Dynamic analyzer selection	72
Boosting search result relevance	74
Static boosting at index-time	74
Dynamic boosting at index-time	75
Conditional indexing	76
Summary	79

Chapter 5: Advanced Querying	81
Filtering	81
Creating a filter factory	82
Adding a filter key	83
Establishing a filter definition	85
Enabling the filter for a query	85
Projection	86
Making a query projection-based	87
Converting projection results to an object form	87
Making Lucene fields available for projection	88
Faceted search	89
Discrete facets	90
Range facets	93
Query-time boosting	95
Placing time limits on a query	95
Summary	97
Chapter 6: System Configuration and Index Management	99
Automatic versus manual indexing	99
Individual updates	100
Adds and updates	100
Deletes	101
Mass updates	101
Defragmenting an index	103
Manual optimization	103
Automatic optimization	104
Custom optimizer strategy	105
Choosing an index manager	106
Configuring workers	107
Execution mode	107
Thread pool	108
Buffer queue	108
Selecting and configuring a directory provider	109
Filesystem-based	109
Locking strategy	110
RAM-based	111
Using the Luke utility	112
Summary	116

Chapter 7: Advanced Performance Strategies	117
General tips	117
Running applications in a cluster	118
Simple clusters	118
Master-slave clusters	119
Directory providers	120
Worker backends	120
A working example	121
Sharding Lucene indexes	125
Summary	127
Index	129

Preface

Over the past decade, users have come to expect software to be highly intelligent when searching data. It is no longer enough to simply make searches case-insensitive, look for keywords as substrings, or other such basic SQL tricks.

Today, when a user searches the product catalog on an e-commerce site, he or she expects keywords to be evaluated across all the data points. Whether a term matches the model number of a computer or the ISBN of a book, the search should still find all the possibilities. To help the user sort through a large number of results, the search should be smart enough to somehow rank them by relevance.

A search should be able to parse words and understand how they might be connected. If you search for the word `development`, then the search should somehow understand that this is related to `developer`, even though neither of the words is a substring of the other.


Above all else, a search should be *nice*. When we post something in an online forum and mistake the words "there", "they're", and "their", people might only criticize our grammar. By contrast, a search should simply understand our typos and be cool about it! A search is at its best when it pleasantly surprises us, seeming to understand the real gist of what we're looking for better than we understood it ourselves.

The purpose of this book is to introduce and explore Hibernate Search, a software package for adding modern search functionality to our own custom applications, without having to invent it from scratch. Because coders usually learn best by looking at real code, this book revolves around an example application. We will stick with this application as we progress through the book, fleshing it out as new concepts are introduced in each chapter.

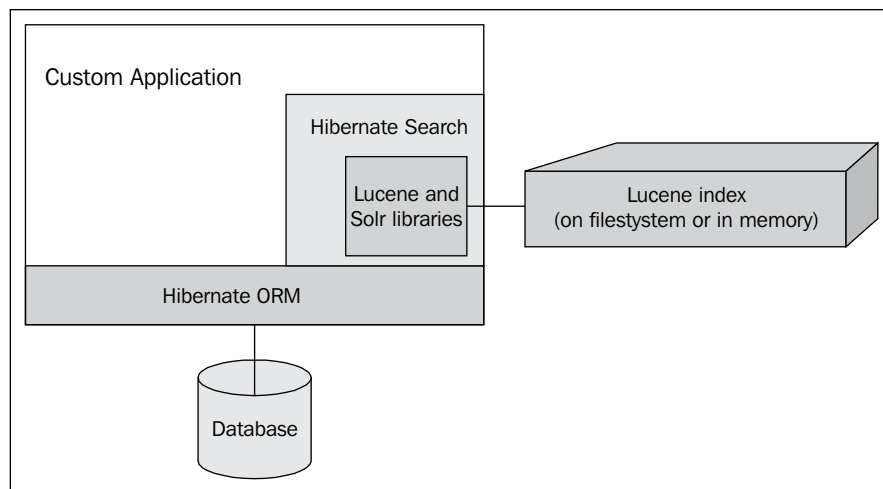
What is Hibernate Search?

The true brain behind this search functionality is Apache Lucene, an open source software library for indexing and searching data. Lucene is an established Java project with a rich history of innovation, although it has been ported to other programming languages as well. It is widely adopted across a variety of industries, with high-profile users ranging from Disney to Twitter.

Lucene is often discussed interchangeably with Apache Solr, a related project. From one perspective, Solr is a standalone search server based on Lucene. However, the dependency relationship can flow both ways. Solr subcomponents are often bundled along with Lucene to enhance its functionality when embedded in other applications.

 Hibernate Search is a thin wrapper around Lucene and optional Solr components. It extends the core Hibernate ORM, the most widely adopted object/relational mapping framework for Java persistence.

The following diagram shows the relationship between all of these components:



Ultimately, Hibernate Search serves two roles:

- First, it translates Hibernate data objects into information that Lucene can use to build search indexes
- Going in the other direction, it translates the results of Lucene searches into a familiar Hibernate format

From a programmer's perspective, he or she is mapping data with Hibernate in the usual way. Search results come back in the same form as normal Hibernate database queries. Hibernate Search hides most of the low-level plumbing with Lucene.

What this book covers

Chapter 1, Your First Application, dives straight away into creating a Hibernate Search application, an online catalog of software apps. We will create one entity class and prepare it for searching, then write a web application to perform searches, and display the results. We will walk through the steps for setting up the application with a server, a database, and a build system, and learn how to go about replacing any of those components with other options.

Chapter 2, Mapping Entity Classes, adds more entity classes to the example application, which are annotated to demonstrate the foundational concepts of Hibernate Search mapping. By the end of this chapter, you will understand how to map the most common entity classes for use with Hibernate Search.

Chapter 3, Performing Queries, expands the example application's queries, to make use of the new mappings. By the end of this chapter, you will understand the most common Hibernate Search query use cases. By this point, the example application will have enough functionality to resemble many production uses of Hibernate Search.

Chapter 4, Advanced Mapping, explains the relationship between Lucene and Solr analyzers, and how to configure an analyzer for more advanced searches. It also covers adjusting a field's weight in the Lucene index, and determines at runtime whether to index an entity at all. By the end of this chapter, you will understand how to fine tune entity indexing. You will have a taste of the Solr analyzer framework, and a grasp of how to explore its functionality on your own. The example application will now support searches that ignore HTML tags, and that find matches for related words.

Chapter 5, Advanced Querying, dives deeper into the querying concepts introduced in *Chapter 3, Performing Queries*, explaining how to get faster performance through projections and results transformation. Faceted searching is explored, as well as an introduction to the native Lucene API. By the end of this chapter, you will have a much more robust understanding of the querying functionality offered by Hibernate Search. The example marketplace application will now use more lightweight, projection-based searches, and have support for organizing the search results by category.

Chapter 6, System Configuration and Index Management, covers Lucene index management, and provides a survey of the advanced configuration options. This chapter dives into some of the more common options in detail, and provides enough background for us to explore others independently. By the end of this chapter, you will be able to perform standard management tasks on the Lucene index used by Hibernate Search, and we will understand the scope of additional functionality available to Hibernate Search through configuration options.

Chapter 7, Advanced Performance Strategies, focuses on improving the runtime performance of Hibernate Search applications, through code as well as server architecture. By the end of this chapter, you will be able to make informed decisions about how to scale a Hibernate Search application as necessary.

What you need for this book

To use the example code covered in this book, you need a computer with a Java Development Kit version 1.6 or higher installed. You should also preferably have Apache Maven installed, or a Java IDE, such as Eclipse, which offers Maven embedded as a plugin.

Who this book is for

The target audience for this book are Java developers who wish to add the search functionality to their applications. The discussion and code examples assume a basic understanding of Java programming. Prior knowledge of **Hibernate ORM**, the **Java Persistence API (JPA 2.0)**, or Apache Maven would be helpful, but is not required.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The `id` field is annotated with both `@Id` and `@GeneratedValue`".

A block of code is set as follows:


```
public App(String name, String image, String description) {
    this.name = name;
    this.image = image;
    this.description = description;
}
```


When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
@Column(length=1000)
@Field
private String description;
```

Any command-line input or output is written as follows:

```
mvn archetype:generate -DgroupId=com.packtpub.hibernate.search.chapter1
-DartifactId=chapter1 -DarchetypeArtifactId=maven-archetype-webapp
```

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Your First Application

To explore the capabilities of **Hibernate Search**, we will work with a twist on the classic "Java Pet Store" sample application. Our version, the "VAPORware Marketplace", will be an online catalog of software apps. Think of such stores run by Apple, Google, Microsoft, Facebook, and... well, pretty much every other company now.

Our app market will give us plenty of opportunities to search data in different ways. Of course, there are titles and descriptions as in most product catalogs. However, software apps involve an expanded set of data points, such as genre, version, and supported devices. These different facets will let us take a look at the many features that Hibernate Search makes available.

At a high level, incorporating Hibernate Search in an application requires the following three steps:

1. Adding information to your entity classes, so that Lucene will know how to index them.
2. Writing one or more search queries in the relevant portions of your application.
3. Setting up your project, so that the required dependencies and configuration for Hibernate Search are available in the first place.

In future projects, after we have a decent understanding of the basics, we would probably start with this third bullet-point. However, for the time being, let us jump straight into some code!

Creating an entity class

To keep things simple, this first cut of our application will include only one entity class. This `App` class describes a software application and is the central entity with which all the other entity classes will be associated. For now though, we will give an "app" three basic data points:

- A name
- An image to display on the marketplace site
- A long description

The Java code is as follows:

```
package com.packtpub.hibernatesearch.domain;

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

@Entity
public class App {

    @Id
    @GeneratedValue
    private Long id;

    @Column
    private String name;

    @Column(length=1000)
    private String description;

    @Column
    private String image;

    public App() {}

    public App(String name, String image, String description) {
        this.name = name;
        this.image = image;
        this.description = description;
    }
}
```

```

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public String getImage() {
        return image;
    }
    public void setImage(String image) {
        this.image = image;
    }
}

```

This class is a basic **plain old Java object (POJO)**, just member variables and getter/setter methods for working with them. However, notice the annotations that are highlighted.



If you are accustomed to Hibernate 3.x, note that version 4.x deprecates many of Hibernate's own mapping annotations in favor of their **Java Persistence API (JPA)** 2.0 counterparts. We will discuss JPA further in *Chapter 3, Performing Queries*. For now, simply notice that the JPA annotations here are essentially identical to their native Hibernate counterparts, other than belonging to the `javax.persistence` package.

The class itself is annotated with `@Entity`, which tells Hibernate to map the class to a database table. Since we did not explicitly specify a table name, by default Hibernate will create a table named `APP` for the `App` class.

The `id` field is annotated with both `@Id` and `@GeneratedValue`. The former simply tells Hibernate that this field maps to the primary key of the database table. The latter declares that the values should be generated automatically when new rows are inserted. This is why our constructor method doesn't populate a value for `id`, because we're counting on Hibernate to handle it for us.

Finally, we annotate our three data points with `@Column`, telling Hibernate that these variables correspond with columns in the database table. Normally, the name of the column will be the same as the variable name, and Hibernate will assume some sensible defaults about the column length, whether to allow null values, and so on. However, these settings may be declared explicitly (as we are doing here), by setting the column length for `description` to 1,000 characters.

Preparing the entity for Hibernate Search

Now that Hibernate knows about our domain object, we need to tell the Hibernate Search add-on how to manage it with **Lucene**.

We can use some advanced options to leverage the full power of Lucene, and as this application develops we will do just that. However, using Hibernate Search in a basic scenario is as simple as adding two annotations.

First, we'll add the `@Indexed` annotation to the class itself:

```
...
import org.hibernate.search.annotations.Indexed;
...
@Entity
@Indexed
public class App implements Serializable {
...
}
```

This simply declares that Lucene should build and use an index for this entity class. This annotation is optional. When you write a large-scale application, many of its entity classes may not be relevant to searching. Hibernate Search only needs to tell Lucene about those types that will be searchable.

Secondly, we will declare searchable data points with the `@Field` annotation:

```
...
import org.hibernate.search.annotations.Field;
...
@Id
@GeneratedValue
private Long id;
```

```

@Column
@Field
private String name;

@Column(length=1000)
@Field
private String description;

@Column
private String image;
...

```

Notice that we're only applying this annotation to the name and description member variables. We did not annotate `image`, because we don't care about searching for apps by their image filenames. We likewise did not annotate `id`, because you don't exactly need a powerful search engine to find a database table row by its primary key!



Deciding what to annotate is a judgment call. The more entities you annotate for indexing, and the more member variables you annotate as fields, the more rich and powerful your Lucene indexes will be. However, if we annotate superfluous stuff just because we can, then we make Lucene do unnecessary work that can hurt performance.

In *Chapter 7, Advanced Performance Strategies*, we will explore such performance considerations in greater depth. Right now, we're all set to search for apps by name or description.

Loading the test data

For test and demo purposes, we will use an embedded database that should be purged and refreshed each time we start the application. With a Java web application, an easy way to invoke the code at startup time is by using `ServletContextListener`. We simply create a class implementing this interface, and annotate it with `@WebListener`:

```

package com.packtpub.hibernatesearch.util;

import javax.servlet.ServletContextEvent;
import javax.servlet.annotation.WebListener;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

```

```
import org.hibernate.service.ServiceRegistry;
import org.hibernate.service.ServiceRegistryBuilder;
import com.packtpub.hibernatesearch.domain.App;

@WebListener
public class StartupDataLoader implements ServletContextListener {
    /** Wrapped by "openSession()" for thread-safety, and not meant to
        be accessed directly. */
    private static SessionFactory sessionFactory;

    /** Thread-safe helper method for creating Hibernate sessions. */
    public static synchronized Session openSession() {
        if(sessionFactory == null) {
            Configuration configuration = new Configuration();
            configuration.configure();
            ServiceRegistry serviceRegistry = new
                ServiceRegistryBuilder().applySettings(
                    configuration.getProperties()).buildServiceRegistry();
            sessionFactory =
                configuration.buildSessionFactory(serviceRegistry);
        }
        return sessionFactory.openSession();
    }

    /** Code to run when the server starts up. */
    public void contextInitialized(ServletContextEvent event) {
        // TODO: Load some test data into the database
    }

    /** Code to run when the server shuts down. */
    public void contextDestroyed(ServletContextEvent event) {
        if(!sessionFactory.isClosed()) {
            sessionFactory.close();
        }
    }
}
```

The `contextInitialized` method will now be invoked automatically when the server starts up. We will use this method to set up a Hibernate session factory, and populate the database with some test data. The `contextDestroyed` method will likewise be automatically invoked when the server shuts down. We will use this method to explicitly close our session factory when done.

Multiple places within our application will need a simple and thread-safe means for opening connections to the database (that is, Hibernate `Session` objects). So, we also add a public static synchronized method named `openSession()`. This method serves as the thread-safe gatekeeper for creating sessions from a singleton `SessionFactory`.



In more complex applications, you would probably use a dependency-injection framework, such as Spring or CDI. This would be a bit distracting in our small example application, but these frameworks give you a safe mechanism for injecting `SessionFactory` or `Session` objects without having to code it manually.

In fleshing out the `contextInitialized` method, we start by obtaining a Hibernate session and beginning a new transaction:

```
...
Session session = openSession();
session.beginTransaction();
...
App app1 = new App("Test App One", "image.jpg",
    "Insert description here");
session.save(app1);

// Create and persist as many other App objects as you like...
session.getTransaction().commit();
session.close();
...
```

Inside the transaction, we can create all the sample data we want, by instantiating and persisting `App` objects. In the interest of readability, only one object is created here. However, the downloadable source code available at <http://www.packtpub.com> contains a full assortment of test examples.

Writing the search query code

Our VAPORware Marketplace web application will be based on a Servlet 3.0 controller/model class, rendering a JSP/JSTL view. The goal is to make things simple, so that we can focus on the Hibernate Search aspects. After reviewing this example application, it should be easy to adapt the same logic in JSF or Spring MVC, or even newer JVM-based frameworks, such as Play or Grails.

To start, we will write a trivial `index.html` page, containing a text box for users to enter search keywords:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>VAPORware Marketplace</title>
</head>
<body>
  <h1>Welcome to the VAPORware Marketplace</h1>
  Please enter keywords to search:
  <form action="search" method="post">
    <div id="search">
      <div>
        <input type="text" name="searchString" />
        <input type="submit" value="Search" />
      </div>
    </div>
  </form>
</body>
</html>
```

This form collects one or more keywords in the CGI parameter `searchString`, and posts it to a URL with the relative `/search` path. We now need to register a controller servlet to respond to those posts:

```
package com.packtpub.hibernatesearch.servlet;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/search")
public class SearchServlet extends HttpServlet {
```

```

protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {

    // TODO: Process the search, and place its results on
    // the "request" object

    // Pass the request object to the JSP/JSTL view
    // for rendering
    getServletContext().getRequestDispatcher(
        "/WEB-INF/pages/search.jsp").forward(request, response);
}

protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
    this.doPost(request, response);
}
}

```

The `@WebServlet` annotation maps this servlet to the relative URL `/search`, so that forms posting to this URL will invoke the `doPost` method. This method will process a search, and forward the request to a JSP view for rendering.

Now, we get to the real heart of the matter—executing the search query. We create a `FullTextSession` object, a Hibernate Search extension that wraps a normal `Session` with Lucene search capability.

```

...
import org.hibernate.Session;
import org.hibernate.search.FullTextSession;
import org.hibernate.search.Search;
...
Session session = StartupDataLoader.openSession();
FullTextSession fullTextSession =
    Search.getFullTextSession(session);
fullTextSession.beginTransaction();
...

```

Now that we have a Hibernate Search session at our disposal, we can grab the user's keyword(s) and perform the Lucene search:

```

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

```

```
String searchString = request.getParameter("searchString");

QueryBuilder queryBuilder =
    fullTextSession.getSearchFactory()
        .buildQueryBuilder().forEntity( App.class ).get();
org.apache.lucene.search.Query luceneQuery =
    queryBuilder
        .keyword()
        .onFields("name", "description")
        .matching(searchString)
        .createQuery();
...
```

As its name suggests, `QueryBuilder` is used to build queries involving a particular entity class. Here, we instantiate a builder for our `App` entity.

Notice the long chain of method calls on the third line of the preceding code. From the perspective Java, we are calling a method, calling another method on the object returned, and repeating that process. However, from a plain English perspective, this chain of method calls resembles a sentence:

*Build a query of **keyword** type, on the entity **fields** "name" and "description", **matching** against the keywords in "searchString".*

This API style is quite intentional. Since it resembles a language in its own right, it is referred to as the Hibernate Search **DSL (domain-specific language)**. If you have ever used criteria queries in Hibernate ORM, then the look-and-feel here should be quite familiar to you.

We have now created an `org.apache.lucene.search.Query` object, which Hibernate Search translates under the covers into a Lucene search. This magic flows in both directions! Lucene search results can be translated into a standard `org.hibernate.Query` object, and used the same as any normal database query:

```
...
org.hibernate.Query hibernateQuery =
    fullTextSession.createFullTextQuery(luceneQuery, App.class);
List<App> apps = hibernateQuery.list();
request.setAttribute("apps", apps);
...
```

Using the `hibernateQuery` object, we fetch all of the `App` entities that were found in our search, and stick them on the servlet request. If you recall, the last line of our method forwards this request to a `search.jsp` view for display.

This JSP view will start off very basic, using JSTL tags to grab the App results off the request and iterate through them:

```
<%@ page language="java" contentType="text/html;
    charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
    <title>VAPORware Marketplace</title>
</head>
<body>
    <h1>Search Results</h1>
    <table>
    <tr>
        <td><b>Name:</b></td>
        <td><b>Description:</b></td>
    </tr>
    <c:forEach var="app" items="${apps}">
    <tr>
        <td>${app.name}</td>
        <td>${app.description}</td>
    </tr>
    </c:forEach>
    </table>
</body>
</html>
```

Selecting a build system

So far, we have approached our application in somewhat reverse order. We basically skipped past the initial project setup and dove straight away into code, so that all the plumbing would make more sense once we got there.

Well, we have now arrived! We need to pull all of this code together into an organized project structure, make sure that all of its JAR file dependencies are available, and establish a process for running the web application or packaging it up as a WAR file. We need a project build system.

One approach that we won't consider is doing all of this by hand. For a small application using bare-bones Hibernate ORM, we might depend on just over a half-dozen JAR files. At that scale, we might consider setting up a standard project in our preferred IDE (for example, Eclipse, NetBeans, or IntelliJ). We could grab a binary distribution from the Hibernate website and copy the necessary JAR files manually, letting the IDE take it from there.

The problem is that Hibernate Search has a lot going on beneath the covers. By the time the time you finish adding the dependencies for Lucene and even the minimal Solr components, the list of dependencies will be multiplied several times over. Even here in the first chapter, our very basic VAPORware Marketplace application already requires over three dozen JAR files to compile and run. These libraries are highly interdependent, and if you upgrade one of them, it can be a real nightmare to avoid conflicts.

At this level of dependency management, it becomes crucial to use an automated build system for sorting out these matters. Throughout the code examples in the book, we will primarily be using Apache Maven for build automation.

The two primary characteristics of Maven are a convention-over-configuration approach to basic builds, and a powerful system for managing a project's JAR file dependencies. As long as a project conforms to a standard structure, we don't even have to tell Maven how to compile it. This is considered boilerplate information. Also, when we tell Maven which libraries and versions a project depends on, Maven will figure out the entire dependency hierarchy for us. It determines which libraries the dependencies themselves depend on, and so forth. A standard repository format has been created for Maven (see <http://search.maven.org> for the largest public example), so that common libraries can all be retrieved automatically without having to hunt them down.

Maven does have its critics. By default, its configuration is XML-based, which has fallen out of fashion in recent years. More importantly, there is a learning curve when a developer needs to do something beyond the boilerplate basics. He or she must learn about the available plugins, how the lifecycle of a Maven build works, and how to configure a plugin for the appropriate lifecycle stage. Many developers have had frustrating experiences with that learning curve.

Several other build systems have been created recently as attempts to harness the same power as Maven in a simpler form (for example, the Groovy-based Gradle, the Scala-based SBT, the Ruby-based Buildr, and so on). However, it is important to note that all of these newer systems are still designed to fetch dependencies from a standard Maven repository. If you wish to use some other dependency management and build system, then the concepts seen in this book will carry over directly to these other tools.

To showcase a more manual non-Maven approach, the sample code available for download from Packt Publishing's website includes an Ant-based version of this chapter's example application. Look for the subdirectory `chapter1-ant`, corresponding to the Maven-based `chapter1` example. A `README` file in the root of this subdirectory highlights the differences. However, the main takeaway is that the concepts shown in the book should translate fairly easily to any modern build system for Java applications.

Setting up the project and importing Hibernate Search

We can create a Maven project using our IDE of choice. Eclipse works with Maven through an optional `m2e` plugin, and NetBeans uses Maven as its native build system out of the box. If Maven is installed on a system, you could also choose to create the project from the command line:

```
mvn archetype:generate -DgroupId=com.packpub.hibernatesearch.chapter1
-DartifactId=chapter1 -DarchetypeArtifactId=maven-archetype-webapp
```

Time can be saved in either case by using a Maven archetype, which is basically a template for a given type of project. Here, `maven-archetype-webapp` gives us an empty web application, configured for packaging as a WAR file. `groupId` and `artifactId` can be anything we wish. They serve to identify our build output if we stored it in a Maven repository.

The `pom.xml` Maven configuration file for our newly-created project starts off looking similar to the following:

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd"
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.packpub.hibernatesearch.chapter1</groupId>
    <artifactId>chapter1</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>
    <name>chapter1</name>
    <url>http://maven.apache.org</url>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <!-- This controls the filename of the built WAR file -->
        <finalName>vaporware</finalName>
    </build>
</project>
```

Our first order of business is to declare which dependencies are needed to compile and run. Inside the `<dependencies>` element, let's add an entry for Hibernate Search:

```
...
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-search</artifactId>
  <version>4.2.0.Final</version>
</dependency>
...
```

Wait, didn't we say earlier that this was going to require over three dozen dependencies? Yes, that is true, but it doesn't mean you have to deal with them all! When Maven reaches out to a repository and grabs this one dependency, it will also receive information about all of its dependencies. Maven climbs down the ladder as deep as it goes, sorting out any conflicts at each step, and calculating a dependency hierarchy so that you don't have to.

Our application needs a database. To keep things simple, we will use H2 (www.h2database.com), an embeddable database system that fits in a single 1 MB JAR file. We will also use **Apache Commons Database Connection Pools** (commons.apache.org/dbcp) to avoid opening and closing database connections unnecessarily. These require declaring only one dependency each:

```
...
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>1.3.168</version>
</dependency>
<dependency>
  <groupId>commons-dbcp</groupId>
  <artifactId>commons-dbcp</artifactId>
  <version>1.4</version>
</dependency>
...
```

Last but not least, we want to specify that our web application is using version 3.x of the JEE Servlet API. In the following dependency, we specify the scope as `provided`, telling Maven not to bundle this JAR inside our WAR file, because we expect our server to make it available anyway:

```
...
<dependency>
  <groupId>javax.servlet</groupId>
```

```
<artifactId>javax.servlet-api</artifactId>
<version>3.0.1</version>
<scope>provided</scope>
</dependency>
...
```

With our POM file complete, we can copy into our project those source files that were created earlier. The three Java classes are listed under the `src/main/java` subdirectory. The `src/main/webapp` subdirectory represents the document root for our web application. The `index.html` search page, and its `search.jsp` results counterpart go here. Download and examine the structure of the project example.

Running the application

Running a Servlet 3.0 application requires Java 6 or higher, and a compatible servlet container such as Tomcat 7. However, if you are using an embedded database to make testing and demonstration easier, then why not use an embedded application server too?

The **Jetty web server** (www.eclipse.org/jetty) has a very nice plugin for Maven and Ant, which let developers launch their applications from a build script without having a server installed. Jetty 8 or higher supports the Servlet 3.0 specification.

To add the Jetty plugin to your Maven POM, insert a small block of XML just inside the root element:

```
<project>
...
<build>
  <finalName>vaporware</finalName>
  <plugins>
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>jetty-maven-plugin</artifactId>
      <version>8.1.7.v20120910</version>
      <configuration>
        <webAppConfig>
          <defaultsDescriptor>
            ${basedir}/src/main/webapp/WEB-INF/webdefault.xml
          </defaultsDescriptor>
        </webAppConfig>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```



```
    </plugins>
  </build>
</project>
```

The highlighted `<configuration>` element is optional. On most operating systems, after Maven has launched an embedded Jetty instance, you can make changes and see them take effect immediately without a restart. However, due to issues with how Microsoft Windows handles file locking, you can't always save changes while the Jetty instance is running.

So if you are using Windows and would like the ability to make changes on-the-fly, make your own custom copy of `webdefault.xml` and save it to the location referenced in the preceding snippet. This file can be found by downloading and opening a `jetty-webapp` JAR file in an unzip tool, or by simply downloading this example application from the Packt Publishing website. The trick for Windows users is to locate the `useFileMappedBuffer` parameter, and change its value to `false`.

Now that you have a web server, let's have it create and manage an H2 database for us. When the Jetty plugin starts up, it will automatically look for the file `src/main/webapp/WEB-INF/jetty-env.xml`. Let's create this file and populate it with the following:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD
    Configure//EN" "http://jetty.mortbay.org/configure.dtd">

<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <New id="vaporwareDB" class="org.eclipse.jetty.plus.jndi.Resource">
    <Arg></Arg>
    <Arg>jdbc/vaporwareDB</Arg>
    <Arg>
      <New class="org.apache.commons.dbcp.BasicDataSource">
        <Set name="driverClassName">org.h2.Driver</Set>
        <Set name="url">
          jdbc:h2:mem:vaporware;DB_CLOSE_DELAY=-1
        </Set>
      </New>
    </Arg>
  </New>
</Configure>
```

This causes Jetty to spawn a pool of H2 database connections, with the JDBC URL specifying an in-memory database rather than a persistent database on the filesystem. We register this data source with the JNDI as `jdbc/vaporwareDB`, so our application can access it by that name. We add a corresponding reference to our application's `src/main/webapp/WEB-INF/web.xml` file:

```
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd" >
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">
  <display-name>VAPORware Marketplace</display-name>
  <resource-ref>
    <res-ref-name>jdbc/vaporwareDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</web-app>
```

Finally, we need to tie this database resource to Hibernate by way of a standard `hibernate.cfg.xml` file, which we will create under `src/main/resources`:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-
    3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.datasource">
      jdbc/vaporwareDB
    </property>
    <property name="hibernate.dialect">
      org.hibernate.dialect.H2Dialect
    </property>
    <property name="hibernate.hbm2ddl.auto">
      update
    </property>
    <property name="hibernate.show_sql">
      false
    </property>
    <property name="hibernate.search.default.directory_provider">
      filesystem
    </property>
```

```
<property name="hibernate.search.default.indexBase">
    target/luceneIndex
</property>

<mapping class=
    "com.packtpub.hibernatesearch.domain.App"/>
</session-factory>
</hibernate-configuration>
```

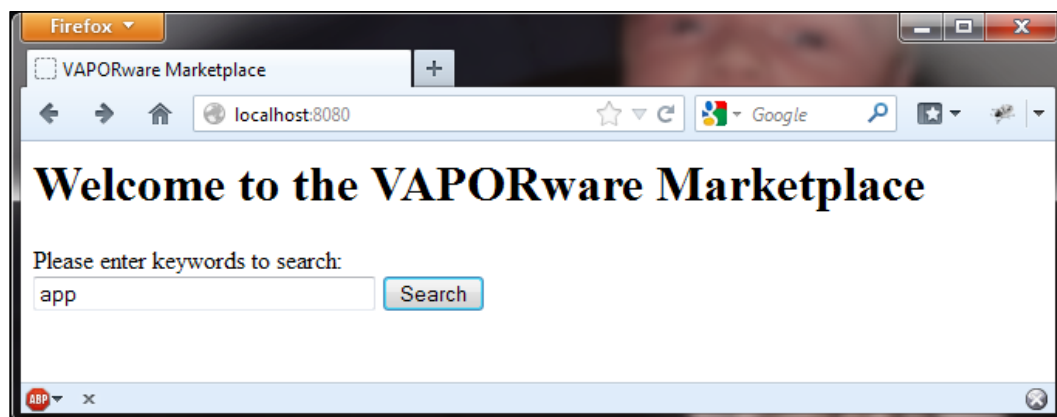
The first highlighted line associates the Hibernate session factory with the Jetty-managed `jdbc/vaporwareDBdata` source. The very last highlighted line declares `App` as an entity class tied to this session factory. Right now we only have this one entity, but we will add more `<class>` elements here as more entities are introduced in later chapters.

In between, most of the `<properties>` elements relate to core settings that are probably familiar to experienced Hibernate users. However, the highlighted properties are directed at the Hibernate Search add-on. `hibernate.search.default.directory_provider` declares that we want to store our Lucene indexes on the filesystem, as opposed to in-memory. `hibernate.search.default.indexBase` specifies a location for the indexes, in a subdirectory within our project that Maven cleans up for us during the build process anyway.

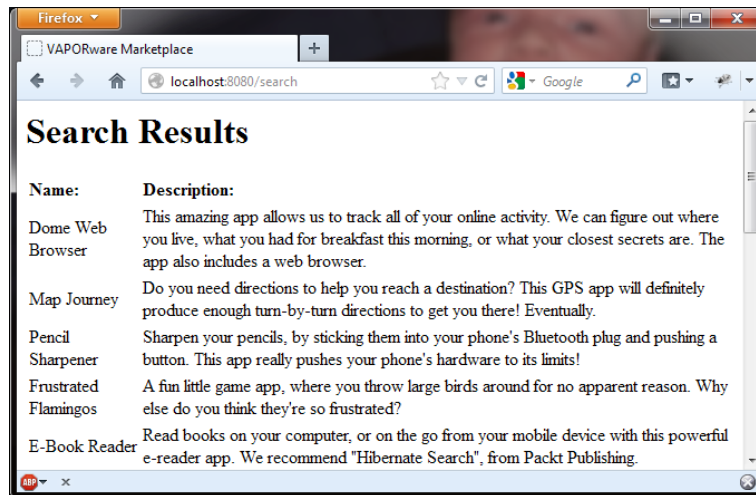
Okay, we have an application, a database, and a server bringing the two together. Now, we can actually deploy and launch, by running Maven with the `jetty:run` goal:

```
mvn clean jetty:run
```

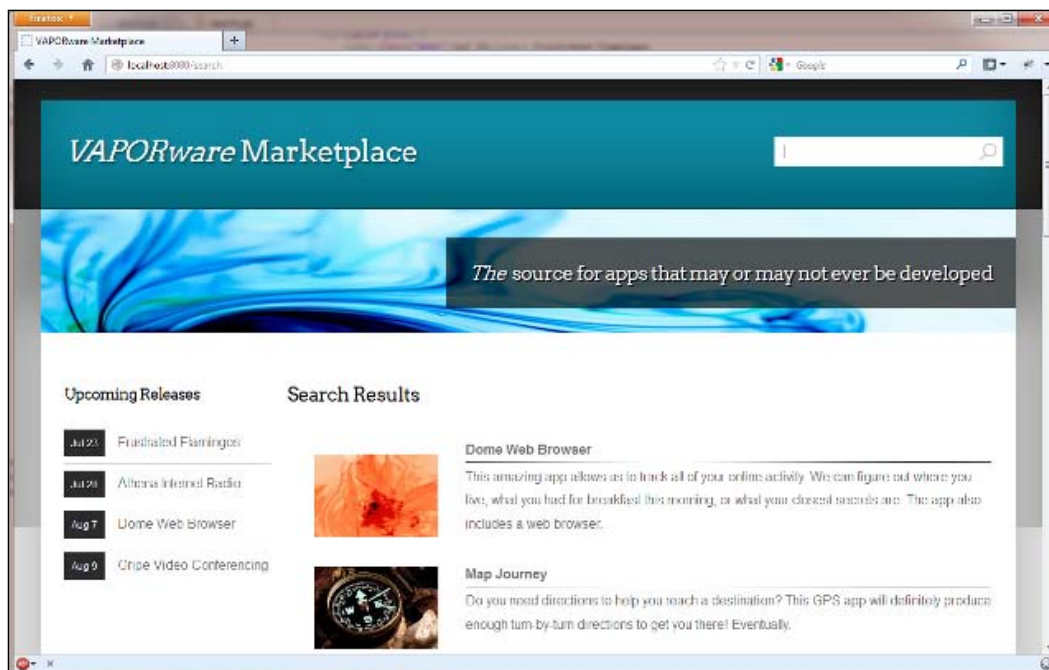
The `clean` goal removes traces of previous builds, and Maven then assembles our web application because this is implied by `jetty:run`. Our code is quickly compiled, and a Jetty server is launched on `localhost:8080`:



We are live! We can now search for apps, using any keywords we like. A quick hint: in the downloadable sample code, all of the test data records contain the word *app* in their descriptions:



The downloadable sample code spruces up the HTML for a more professional look. It also adds each app's image alongside its name and description:



The Maven command `mvn clean package` lets us package the application up as a WAR file, so we can deploy it to a standalone server outside of the Maven Jetty plugin. You can use any Java server compatible with the Servlet 3.0 specification (for example, Tomcat 7+), so long as you know how to set up a data source with the JNDI name `jdbc/vaporwareDB`.

For that matter, you can replace H2 with any standalone database that you like. Just add an appropriate JDBC driver to your Maven dependencies, and update the settings within `persistence.xml`.

Summary

In this chapter, we learned about the relationship between Hibernate ORM, the Hibernate Search add-on, and the underlying Lucene search engine. We saw how to map entities and fields to make them available for searching. We used the Hibernate Search DSL to write a full-text search query across multiple fields, and worked with the results as we would during a normal database query. We used an automated build process to compile our application, and deployed it to a web server with a live database.

With these tools alone, we could incorporate Hibernate Search right now into many real-world applications, using any other server or database. In the next chapter, we will dive deeper into the options that Hibernate Search makes available for mapping entity objects to Lucene indexes. We will see how to handle an expanded data model, associating our VAPORware apps with devices and customer reviews.

2

Mapping Entity Classes

In *Chapter 1, Your First Application*, we used core Hibernate ORM to map an entity class to a database table, and then we used Hibernate Search to map two of its fields to a Lucene index. By itself, this provides a lot of search functionality that would be very cumbersome to code from scratch.

However, real-world applications usually involve numerous entities, many of which should be available for searching. Entities may be associated with each other, and our queries need to understand those associations so that we can search across multiple entities at once. We might want to declare that some mappings are more relevant to a search than others, or we might want to skip indexing data under certain conditions.

In this chapter, we will start to dive deeper into the options that Hibernate Search makes available for mapping entities. As a first step, we must take a look at the API options available in Hibernate ORM. How we map our entity classes to the database influences how Hibernate Search maps them to Lucene.

Choosing an API for Hibernate ORM

When the Hibernate Search documentation mentions different APIs for Hibernate ORM, it can be confusing. In some cases, this might refer to whether database queries are performed using an `org.hibernate.Session` or a `javax.persistence.EntityManager` object (an important part of the next chapter). However, in the context of entity mapping, this refers to the three different approaches offered by Hibernate ORM:

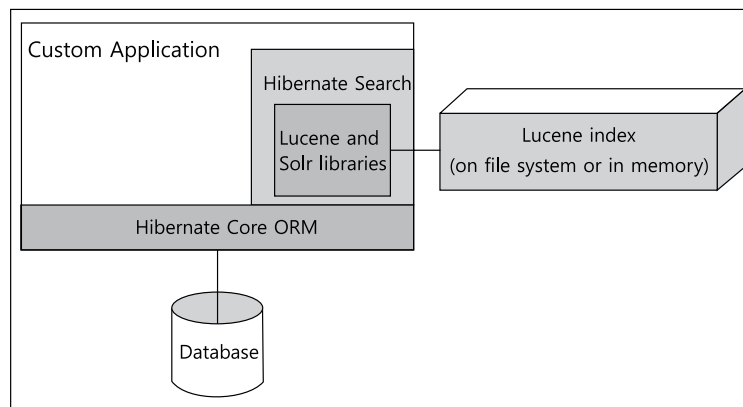
- Annotation-based mapping with classic Hibernate-specific annotations
- Annotation-based mapping with the Java Persistence API (JPA 2.0)
- XML-based mapping with `hbm.xml` files

If you have only used Hibernate ORM with its classic annotations or XML-based mappings, or if you are new to Hibernate altogether, then this may be your first exposure to JPA. In a nutshell, JPA is a specification intended to serve as the official standard for object-relational mapping and similar features.

The idea is to provide for ORM what JDBC provides for low-level database connectivity. Once a developer has learned JDBC, they can quickly work with any database driver that implements the API (for example, Oracle, PostgreSQL, MySQL, and so on). Likewise, if you understand JPA, then you should be able to easily switch between ORM frameworks, such as Hibernate, EclipseLink, and Apache OpenJPA.

In practice, different implementations often have their own quirks and proprietary extensions, which can cause transition headaches. However, a common standard does reduce the pain and learning curve dramatically.

A comparison of using the Hibernate ORM native API versus using JPA for entity mapping is shown in the following diagram:



The good news for long-time Hibernate developers is that JPA entity mapping annotations are remarkably similar to Hibernate's own annotations. In fact, the founder of Hibernate worked on the committee that developed JPA, and these two APIs have strongly influenced each other.

The less-good news, depending on your perspective, is that Hibernate ORM 4.x deprecates its own mapping annotations in favor of their JPA counterparts. Those older annotations are targeted for removal in Hibernate ORM 5.x.

[ It doesn't make sense to write a new code today using this deprecated approach, so we will disregard Hibernate-specific mapping annotations.]

The third option, XML-based mapping, is still commonly found in legacy applications. It is falling out of favor, and the Hibernate Search documentation jokes about XML being unfit for the 21st century! Of course, that is somewhat tongue-in-cheek, considering that basic Hibernate configuration still lives in a `hibernate.cfg.xml` or `persistence.xml` file. Still, the clear trend with most Java frameworks is to use annotations for configuration that is tied to a particular class, and to use some form of text file for global configuration.

Even if you are using `hbm.xml` files to map your entities to the database, you can still use the Hibernate Search annotations to map those entities to Lucene indexes. The two are perfectly compatible. This is convenient if you want to add Hibernate Search to a legacy application with minimal effort, or if you have a philosophical preference for `hbm.xml` files even when developing new applications.

The sample code for this book includes three versions of the VAPORware Marketplace application for this chapter:

- The `chapter2` subdirectory continues where *Chapter 1, Your First Application* left off, using JPA annotations for mapping entities to both the database and Lucene.
- The `chapter2-xml` subdirectory is a variant of the same code, modified to mix XML-based database mapping with JPA-based Lucene mapping.
- The `chapter2-mapping` subdirectory uses a special API to avoid annotations altogether. This is discussed further in the *Programmatic Mapping API* section near the end of this chapter.

You should explore all of this example code in detail to understand the available options. However, unless otherwise noted, the code examples in this book will focus on JPA annotations for both database and Lucene mapping.



When JPA annotations are used for database mapping, Hibernate Search automatically creates a Lucene identifier for fields annotated with `@Id`.

For whatever reason, Hibernate Search cannot do the same with Hibernate ORM's own mapping API. So when you are not using JPA to map entities to the database, you must also add the `@DocumentId` annotation to fields that should be used as Lucene identifiers (entities are known as **documents** in Lucene terminology).

Field mapping options

In *Chapter 1, Your First Application*, we saw that member variables on a Hibernate-managed class are made searchable with the `@Field` annotation. Hibernate Search will put information about annotated fields into one or more Lucene indexes, using some sensible defaults.

However, you can customize indexing behavior in numerous ways, some of which are optional elements in the `@Field` annotation itself. Most of these elements will be explored further throughout this book, but we will briefly introduce them here in one centralized list:

- `analyze`: This tells Lucene whether to store the field's data as is, or put it through analysis, parsing, and processing it in various ways. It can be set to `Analyze.YES` (the default) or `Analyze.NO`. We will see this again in *Chapter 3, Performing Queries*.
- `index`: This controls whether or not the field should be indexed by Lucene. It can be set to `Index.YES` (the default) or `Index.NO`. It may sound nonsensical to use the `@Field` annotation and then not index the field, but this will make more sense after seeing projection-based searches in *Chapter 5, Advanced Querying*.
- `indexNullAs`: This declares how to handle null field values. By default, null values will simply be ignored and excluded from Lucene indexes. However, with this element fully covered in *Chapter 4, Advanced Mapping*, you can force null fields to be indexed with some default value instead.
- `name`: This is a custom name, describing this field in the Lucene indexes. By default, Hibernate Search will use the name of the annotated member variable.
- `norms`: This determines whether or not to store index-time information used for boosting, or adjusting the default relevance of search results. It can be set to `Norms.YES` (the default) or `Norms.NO`. Index-time boosting will appear in *Chapter 4, Advanced Mapping*.
- `store`: Normally, fields are indexed in a manner optimized for searching, but this might not make it possible to retrieve the data in its original form. This option causes the raw data to be stored in such a way that you can later retrieve it directly from Lucene, rather than the database. It can be set to `Store.NO` (the default), `Store.YES`, or `Store.COMPRESS`. We will use this with projection-based searches in *Chapter 5, Advanced Querying*.

Multiple mappings for the same field

Sometimes, you need to use one set of options to do certain things with a field, and other set of options to do other things. We will see this later in *Chapter 3, Performing Queries* when we make a field both searchable and sortable.

For the time being, suffice it to say that you can have as many custom mappings as you wish for the same field. Just include multiple `@Field` annotations, wrapped within the plural `@Fields`:

```
...
@Column
@Fields({
    @Field,
    @Field(name="sorting_name", analyze=Analyze.NO)
})
private String name;
...
```

Don't worry too much about this example right now. Just note that when you create more than one mapping for the same field, you need to give them distinct names with the `name` element, so that you can later reference the correct mapping.

Mapping numeric fields

In *Chapter 1, Your First Application*, our entity mapping examples dealt exclusively with string properties. It is likewise perfectly fine to use the same `@Field` annotation with other basic data types as well.

However, fields mapped this way are indexed by Lucene in string format. That is very inefficient for techniques that we will explore later, such as sorting and querying over a range.

To improve the performance of such operations, Hibernate Search offers a specialized data structure for indexing numeric fields. This option is available when mapping fields of type `Integer`, `Long`, `Float`, and `Double` (or their primitive counterparts).

To use this optimized data structure for a numeric field, you simply add the `@NumericField` annotation in addition to the normal `@Field`. As an example, let's give the `App` entity in our VAPORware Marketplace application a field for price:

```
...
@Column
@Field
@NumericField
private float price;
...
```

If you are applying this annotation to a property that has been mapped multiple times with `@Fields`, you must specify *which* of those mappings should use the specialized data structure. This is done by giving the `@NumericField` annotation an optional `forField` element, set to the same name as the desired `@Field`.

Relationships between entities

Every time an entity class is annotated with `@Indexed`, by default Hibernate Search will create a Lucene index just for that class. We can have as many entities, and as many separate indexes, as we wish. However, searching each index separately would be a very awkward and cumbersome approach.

Most Hibernate ORM data models already capture the various associations between entity classes. When we search an entity's Lucene index, shouldn't Hibernate Search follow those associations? In this section we will see how to make it do just that.

Associated entities

So far, the entity fields in our example application have been simple data types. The `App` class represents a table named `APP`, and its member variables map to columns in that table. Now let's add a complex type field, for a second database table that is associated through a foreign key.

Online app stores usually support a range of different hardware devices. So we will create a new entity called `Device`, representing devices for which an `App` entity is available.

```
@Entity
public class Device {

    @Id
    @GeneratedValue
    private Long id;

    @Column
    @Field
    private String manufacturer;

    @Column
    @Field
    private String name;

    @ManyToMany(mappedBy="supportedDevices",
```

```

        fetch=FetchType.EAGER,
        cascade = { CascadeType.ALL }
    )
    @ContainedIn
    private Set<App> supportedApps;

    public Device() {
    }

    public Device(String manufacturer, String name,
        Set<App>supportedApps) {
        this.manufacturer = manufacturer;
        this.name = name;
        this.supportedApps = supportedApps;
    }

    //
    // Getters and setters for all fields...
    //
}

```

Most details of this class should be familiar from *Chapter 1, Your First Application*. `Device` is annotated with `@Entity`, so Hibernate Search will create a Lucene index just for it. The entity class contains searchable fields for device name, and manufacturer name.

However, the `supportedApps` member variable introduces a new annotation, for making the association between these two entities bidirectional. An `App` entity will contain a list of all its supported devices, and a `Device` entity will contain a list of all its supported apps.



If for no other reason, using bidirectional associations improves the reliability of Hibernate Search.

A Lucene index contains denormalized data from associated entities, but those entities are still primarily tied to their own Lucene indexes. To cut a long story short, when the association between two entities is bidirectional, and changes are set to cascade, then you can count on both indexes being updated when either entity changes.

The Hibernate ORM reference manual describes several bidirectional mapping types and options. Here we are using `@ManyToMany`, to declare a many-to-many relationship between the `App` and `Device` entities. The `cascade` element is set to ensure that changes on this end of the association properly update the other side.



Normally, Hibernate is "lazy". It doesn't actually fetch associated entities from the database until they are needed.

However, here we are writing a multi-tiered application, and the controller servlet has already closed the Hibernate session by the time our search results JSP receives these entities. If a view tries to fetch associations after the session has closed, errors will occur.

There are several ways around this problem. For simplicity, we are also adding a `fetch` element to the `@ManyToMany` annotation, changing the fetch type from "lazy" to "eager". Now when we retrieve a `Device` entity, Hibernate will immediately fetch all the associated `App` entities while the session is still open.

Eager fetching is very inefficient with large amounts of data, however, so in *Chapter 5, Advanced Querying*, we will explore a more advanced strategy for handling this.

Everything about `supportedApps` discussed so far has been in the realm of Hibernate ORM. So last but not least, we will add the Hibernate Search `@ContainedIn` annotation, declaring that `App`'s Lucene index should contain data from `Device`. Hibernate ORM already saw these two entities as being associated. The Hibernate Search `@ContainedIn` annotation sets up a bidirectional association from Lucene's perspective too.

The other half of the bidirectional association involves giving the `App` entity class a list of supported `Device` entity classes.

```
...
@ManyToMany(fetch=FetchType.EAGER, cascade = { CascadeType.ALL })
@IndexedEmbedded(depth=1)
private Set<Device>supportedDevices;
...
// Getter and setter methods
...
```

This is very similar to the `Device` side of the association, except that the `@IndexedEmbedded` annotation here serves as the counterpoint to `@ContainedIn`.



If your associated objects contain other associated objects themselves, then you might end up indexing a lot more data than you wanted. Even worse, you could run into problems with circular dependencies.

To safeguard against this, set the `@IndexEmbedded` annotation's optional `depth` element to a max limit. When indexing objects, Hibernate Search will go no further than the specified number of levels.

The previous code specifies a depth of one level. This means that an app will be indexed with information about its supported devices, but *not* information about a device's other supported apps.

Querying associated entities

Once associated entities have been mapped for Hibernate Search, it is easy to include them in search queries. The following code snippet updates `SearchServlet` to add `supportedDevices` to the list of fields searched:

```
...
QueryBuilderQueryBuilder =
    fullTextSession.getSearchFactory().buildQueryBuilder()
        .forEntity(App.class).get();
org.apache.lucene.search.Query luceneQuery = queryBuilder
    .keyword()
    .onFields("name", "description", "supportedDevices.name")
    .matching(searchString)
    .createQuery();
org.hibernate.Query hibernateQuery =
    fullTextSession.createFullTextQuery(luceneQuery, App.class);
...
```

Complex types are a bit different from the simple data types we have worked with so far. With complex types, we are not really interested in the field itself, because the field is actually just an object reference (or a collection of object references).

What we really want our searches to match are the simple data type fields within the complex type. In other words, we want to search the `Device` entity's `name` field. So, as long as an associated class field has been indexed (that is, with the `@Field` annotation), it can be queried with a `[entity field].[nested field]` format, such as `supportedDevices.name` in the previous code.

In the sample code for this chapter, `StartupDataLoader` has been expanded to save some `Device` entities in the database, and associate them with the `App` entities. One of these test devices is named `xPhone`. When we run the VAPORware Marketplace application and search for this keyword, the search results will include apps that are compatible with the `xPhone`, even if that keyword doesn't appear in the name or description of the app itself.

Embedded objects

Associated entities are full-blown entities in their own right. They typically correspond to a database table and Lucene index of their own, and may stand apart from their associations. For example, if we delete an app entity that is supported on the `xPhone`, that doesn't mean we want to delete the `xPhone Device` too.

There is a different type of association, in which the lifecycle of associated objects depends on the entity that contains them. If the VAPORware Marketplace apps had customer reviews, and an app was permanently deleted from the database, then we would probably expect all its customer reviews to be removed along with it.



Classic Hibernate ORM terminology refers to such objects as **components** (or sometimes **elements**). In the newer JPA jargon, they are known as **embedded objects**.

Embedded objects are not entities themselves. Hibernate Search does not create separate Lucene indexes for them, and they cannot be searched outside the context of the entity containing them. Otherwise, they look and feel quite similar to associated entities.

Let's give the example application an embedded object type for customer reviews. A `CustomerReview` instance will consist of the username of the person submitting the review, the rating they gave (for example, five stars), and any additional comments they wrote.

```
@Embeddable
public class CustomerReview {

    @Field
    private String username;

    private int stars;

    @Field
    private String comments;

    public CustomerReview() {
```

```

    }

    public CustomerReview(String username,
        int stars, String comments) {
        this.username = username;
        this.stars = stars;
        this.comments = comments;
    }

    // Getter and setter methods...
}

```

This class is annotated with `@Embeddable` rather than the usual `@Entity` annotation, telling Hibernate ORM that the lifecycle of a `CustomerReview` instance is dependent on whichever entity object contains it.

The `@Field` annotation is applied to searchable fields as before. However, Hibernate Search will not create a standalone Lucene index just for `CustomerReview`. This annotation only adds information to the indexes of other entities that contain this embeddable class.

In our case, the containing class will be `App`. Let's give it a set of customer reviews as a member variable:

```

...
@ElementCollection(fetch=FetchType.EAGER)
@Fetch(FetchMode.SELECT)
@IndexedEmbedded(depth=1)
private Set<CustomerReview>customerReviews;
...

```

Rather than one of the usual JPA relationship annotations (for example, `@OneToOne`, `@ManyToMany`, and so on), this field is annotated as a JPA `@ElementCollection`. If this field were a single object, no annotation would be necessary. JPA would simply figure it out based on that object class having the `@Embeddable` annotation. However, the `@ElementCollection` annotation is necessary when dealing with collections of embeddable elements.



When using classic XML-based Hibernate mapping, the `hbm.xml` file equivalents are `<component>` for single instances, and `<composite-element>` for collections. See the `chapter2-xml` variant of the downloadable sample application source.

The `@ElementCollection` annotation has a `fetch` element set to use eager fetching, for the same reasons discussed earlier in this chapter.

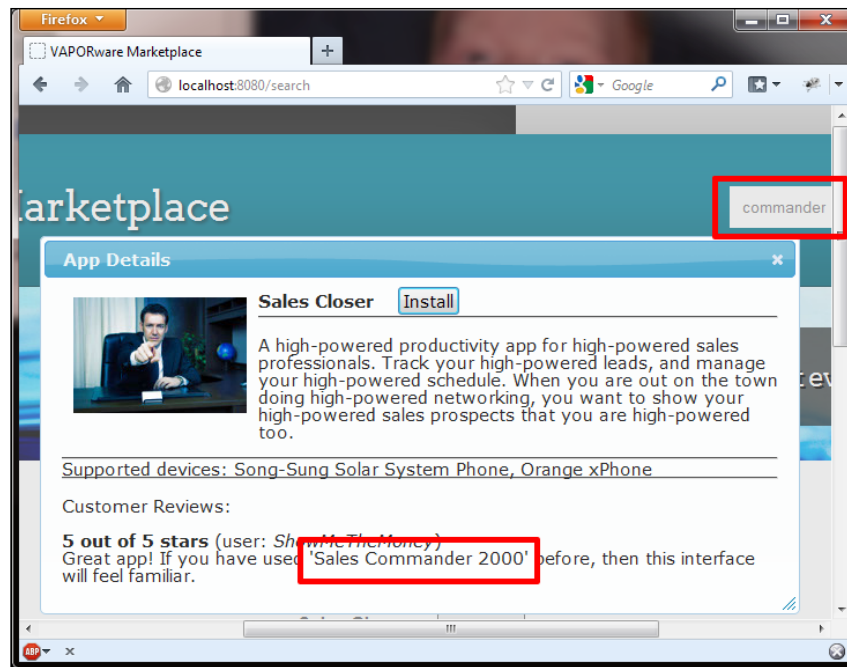
On the next line we use the Hibernate-specific `@Fetch` annotation, to ensure that the `CustomerReview` instances are fetched through multiple `SELECT` statements rather than a single `OUTER JOIN`. This avoids duplication of customer reviews, due to Hibernate ORM quirks that are discussed further in comments within the downloadable source code. Unfortunately, this mode is inefficient when dealing with very large collections, so you may wish to consider another approach in such cases.

Querying embedded objects is the same as with associated entities. Here is the query code snippet from `SearchServlet`, modified to also search against the comments fields of the embedded `CustomerReview` instances:

```
...
QueryBuilderQueryBuilder =
fullTextSession.getSearchFactory().buildQueryBuilder()
    .forEntity(App.class).get();
org.apache.lucene.search.Query luceneQuery = queryBuilder
    .keyword()
    .onFields("name", "description", "supportedDevices.name",
        "customerReviews.comments")
    .matching(searchString)
    .createQuery();
org.hibernate.Query hibernateQuery = fullTextSession.
createFullTextQuery(
    luceneQuery, App.class);
...
```

Now we have a query that is really doing some searching! The `chapter2` version of `StartupDataLoader` has been extended to load some customer reviews for all of the test apps. Searches will now produce results when a match is found in a customer review, even though the keyword doesn't otherwise appear in the `App` itself.

The HTML in the VAPORware Marketplace application has also been updated. Now each search result has a **Full Details** button, which pops-up a modal box with supported devices and customer reviews for that app. Notice that the search keyword in this screenshot is matched against a customer review rather than the actual app description:



Partial indexing

Associated entities each have their own Lucene index, and also store some data in each other's indexes. With embedded objects, search information is stored *exclusively* in the containing entity's index.

However, bear in mind that these classes may be associated or embedded in more than one place. For example, if you had the `Customer` and `Publisher` entities in your data model, both of them might have an embedded object of type `Address`.

Normally, we use the `@Field` annotation to tell Hibernate Search which fields should be indexed and searchable. However, what if we want this to vary with associated or embedded objects? What if we want a field to be indexed, or not indexed, depending on which other entity contains it? Hibernate Search provides this ability through an optional element in the `@IndexedEmbedded` annotation. This `includePaths` element indicates that within the Lucene index for *this* containing entity, only certain fields of the associated entity or embedded object should be included.

In our example application, the `CustomerReview` class has both its `username` and `comments` variable annotated as searchable fields. However, let's say that for the `customerReviews` embedded within `App`, we only care about searching on `comments`. The change to `App` looks like this:

```
...
@ElementCollection(fetch=FetchType.EAGER)
@Fetch(FetchMode.SELECT)
@IndexedEmbedded(depth=1, includePaths = { "comments" })
private Set<CustomerReview>customerReviews;
...
```

Even though `CustomerReview.username` is annotated with `@Field`, that field will not be added to the Lucene index for `App`. This saves space, and improves performance by not making Lucene work hard on unnecessary indexing. The only trade-off is that to prevent errors, we must remember to avoid using any non-included fields in our query code.

The programmatic mapping API

In the beginning of this chapter we said that even when you map entities to the database with `hbm.xml` files, you can still use Hibernate Search annotations mapping to Lucene. However, if you really want to avoid putting annotations in your entity classes altogether, there is an API available for declaring your Lucene mappings programmatically at runtime.

This might come in handy if your search configuration needs to change at runtime based on some circumstances. It is also the only approach available if you cannot alter your entity classes for some reason, or if you are a hard-line believer in separating your configuration from your POJO's.

The heart of the programmatic mapping API is the `SearchMapping` class, which stores the Hibernate Search configuration that is normally pulled from annotations. Typical usage looks like the query DSL code that we saw in the previous chapter. You call a method on a `SearchMapping` object, call a method on the object returned, and so on in a long nested series.

The methods available at each step of the way intuitively resemble the search annotations that you have already seen. The `entity()` method replaces the `@Entity` annotation, `indexed()` replaces `@Indexed`, `field()` replaces `@Field`, and so on.



If you need to use the programmatic mapping API in an application, then you can find more details in *Reference Manual* and *Javadocs*, both available at <http://www.hibernate.org/subprojects/search/docs>.

The starting point in Javadocs is the `org.hibernate.search.cfg.SearchMapping` class, and the other relevant classes are all in the `org.hibernate.search.cfg` package as well.

In the downloadable source code available from the Packt Publishing website, the `chapter2-mapping` subdirectory contains a version of the VAPORware Marketplace application that uses the programmatic mapping API.

This version of the example application includes a factory class, with a method that configures and returns a `SearchMapping` object upon demand. It doesn't matter what you name the class or the method, so long as the method is annotated with `@org.hibernate.search.annotations.Factory`:

```
public class SearchMappingFactory {

    @Factory
    public SearchMapping getSearchMapping() {

        SearchMapping searchMapping = new SearchMapping();

        searchMapping
            .entity(App.class)
                .indexed()
                .interceptor(IndexWhenActiveInterceptor.class)
                .property("id", ElementType.METHOD).documentId()
                .property("name", ElementType.METHOD).field()
                .property("description", ElementType.METHOD).field()
                .property("supportedDevices",
                    ElementType.METHOD).indexEmbedded().depth(1)
                .property("customerReviews",
                    ElementType.METHOD).indexEmbedded().depth(1)

            .entity(Device.class)
                .property("manufacturer", ElementType.METHOD).field()
                .property("name", ElementType.METHOD).field()
                .property("supportedApps",
                    ElementType.METHOD).containedIn()
    }
}
```

```
        .entity(CustomerReview.class)
        .property("stars", ElementType.METHOD).field()
        .property("comments", ElementType.METHOD).field();

    return searchMapping;
}

}
```

Notice that this factory method is only three lines long, strictly speaking. The bulk of it is one continuous line of chained method calls, originating from the `SearchMapping` object, that map our three persistent classes.

To integrate the mapping factory into Hibernate Search, we add a property to the main `hibernate.cfg.xml` configuration file:

```
...
<property name="hibernate.search.model_mapping">
    com.packtpub.hibernatesearch.util.SearchMappingFactory
</property>
...
```

Now, whenever Hibernate ORM opens a `Session`, Hibernate Search and all of the Lucene mappings come along for the ride!

Summary

In this chapter, we expanded our knowledge of how to map classes for searching. We can now use Hibernate Search to map entities and other classes to Lucene, regardless of how Hibernate ORM maps them to the database. If we ever need to map classes to Lucene without adding annotations, we can use a programmatic mapping API to handle this at runtime.

We have now seen how to manage Hibernate Search across associated entities, as well as embedded objects whose lifecycle depend on their containing entity. In both cases, we covered some obscure quirks that can trip up developers. Finally, we learned how to control which fields of an associated or embedded class are indexed, depending on which entity contains them.

In the next chapter, we will use these mappings in a variety of search query types, and explore some important features common to all of them.

3

Performing Queries

In the previous chapter, we created various types of persistent objects and mapped them to Lucene search indexes in various ways. However, we have basically used the same keyword query in all the versions of the example application so far.

In this chapter, we will explore other search query types offered by the Hibernate Search DSL, as well as important features such as sorting and pagination that are common to all of them.

Mapping API versus query API

So far, we discussed API alternatives for mapping classes to the database with Hibernate ORM. You can map your classes with XML or annotations, using JPA or the traditional API, and Hibernate Search will work fine so long as you are aware of some minor differences.

However, when we talk about which API a Hibernate application uses, there are two parts to the answer. Not only is there more than one approach for mapping classes to the database, there are also options for how to query the database at runtime. Hibernate ORM has its traditional API, based on the `SessionFactory` and `Session` classes. It also offers an implementation of the corresponding JPA standards, built around `EntityManagerFactory` and `EntityManager`.

You might have noticed that in the sample code so far, we've been mapping classes to the database with JPA annotations and using the traditional Hibernate `Session` class to query them. This may seem confusing at first, but the mapping and the query APIs are essentially interchangeable. You can mix and match!

So which approach should you use in the Hibernate Search projects? There are advantages to sticking with common standards as much as possible. Once you are experienced with JPA, those skills transfer when you work on other projects that use different JPA implementations.

On the other hand, Hibernate ORM's traditional API is more powerful than the generic JPA standards. Also, Hibernate Search is an extension of Hibernate ORM. You can't migrate a project to a different JPA implementation without first finding some other search strategy altogether.



So in a nutshell, there is a strong argument for using JPA standards whenever they are adequate. However, Hibernate Search requires Hibernate ORM anyway, so there's no sense in being too dogmatic. Throughout this book, most of the example code will use JPA annotations for mapping classes, and use the traditional Hibernate Session class for queries.

Using JPA for queries

Although we will focus on the traditional query API, the downloadable source code also contains a different version of the example application in a `chapter3-entitymanager` folder. This VAPORware Marketplace variation demonstrates the use of JPA across the board, for both mapping and queries.

In the search controller servlet, rather than using a Hibernate `SessionFactory` object to create a `Session` object, it uses a JPA `EntityManagerFactory` instance to create an `EntityManager` object:

```
...
// The "com.packtpub.hibernatesearch.jpa" identifier is declared
// in "META-INF/persistence.xml"
EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory(
        "com.packtpub.hibernatesearch.jpa");
EntityManager entityManager =
    entityManagerFactory.createEntityManager();
...
```

We have already seen code samples that use the traditional query API. In those previous examples, the Hibernate ORM `Session` objects were wrapped within Hibernate Search `FullTextSession` objects. These then produced Hibernate Search `FullTextQuery` objects, which implement the `core.org.hibernate.Query` interface:

```
...
FullTextSession fullTextSession = Search.getFullTextSession(session);
...
org.hibernate.search.FullTextQuery hibernateQuery =
    fullTextSession.createFullTextQuery(luceneQuery, App.class);
...
```

Contrast that with JPA, where regular `EntityManager` objects are likewise wrapped by `FullTextEntityManager` objects. These create `FullTextQuery` objects, implementing the standard `javax.persistence.Query` interface:

```
...
FullTextEntityManager fullTextEntityManager =
    org.hibernate.search.jpa.Search.getFullTextEntityManager(
        entityManager);
...
org.hibernate.search.jpa.FullTextQuery jpaQuery =
    fullTextEntityManager.createFullTextQuery(luceneQuery, App.
        class);
...
```

The traditional `FullTextQuery` class and its JPA counterpart are very similar, but they are separate classes imported from different Java packages. Both offer hooks to much of the Hibernate Search functionality that we've seen so far, and will further explore.



Either version of `FullTextQuery` can be cast to its respective query type, although doing so costs you direct access to the Hibernate Search methods. So, be sure to call any extension methods prior to casting.

If you need to access the non-standard methods after casting to a JPA query, then you can use that interface's `unwrap()` method to get back to the underlying `FullTextQuery` implementation.

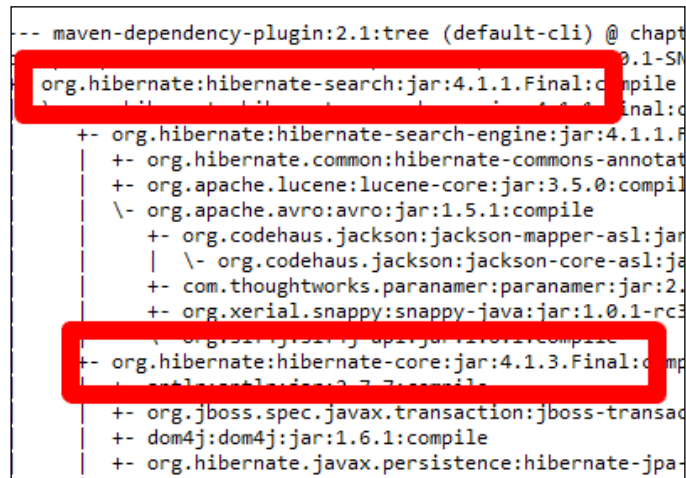
Setting up a project for Hibernate Search and JPA

When your Maven-based project includes the `hibernate-search` dependency, it automatically pulls over three dozen related dependencies for you. Unfortunately, JPA query support is not one of them. To use JPA-style queries, we must declare an extra `hibernate-entitymanager` dependency ourselves.

Its version needs to match the version of `hibernate-core` that is already in the dependency hierarchy. This will not always be in sync with the `hibernate-search` version.

Your IDE may offer a way to present the dependency hierarchy visually. Either way, you can always use Maven from the command line to get the same information with this command:

```
mvn dependency:tree
```



```
-- maven-dependency-plugin:2.1:tree (default-cli) @ chapt
org.hibernate:hibernate-search:jar:4.1.1.Final:compile
+- org.hibernate:hibernate-search-engine:jar:4.1.1.Final:compile
| +- org.hibernate.common:hibernate-commons-annotations:jar:3.2.0.Final:compile
| +- org.apache.lucene:lucene-core:jar:3.5.0:compile
| \- org.apache.avro:avro:jar:1.5.1:compile
|   +- org.codehaus.jackson:jackson-mapper-asl:jar:1.8.8:compile
|   | \- org.codehaus.jackson:jackson-core-asl:jar:1.8.8:compile
|   +- com.thoughtworks.paranamer:paranamer:jar:2.2.1:compile
|   +- org.xerial.snappy:snappy-java:jar:1.0.1-rc3:compile
|   \- org.glassfish.jersey.core:jersey-client:jar:2.7:compile
+- org.hibernate:hibernate-core:jar:4.1.3.Final:compile
| +- org.jboss.spec.javax.transaction:jboss-transaction-api_1.2_spec:jar:1.0.1.Final:compile
| +- dom4j:dom4j:jar:1.6.1:compile
| \- org.hibernate.javax.persistence:hibernate-jpa-2.0-api:jar:1.0.1.Final:compile
```

As shown in this output, Hibernate Search version 4.2.0.Final uses core Hibernate ORM version 4.1.9.Final. So a `hibernate-entitymanager` dependency should be added to the POM, using the same version as core:

```
...
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>4.1.9.Final</version>
</dependency>
...
```

The Hibernate Search DSL

Chapter 1, Your First Application, introduced the Hibernate Search DSL, which is the most straightforward approach for writing search queries. When using the DSL, method calls are chained together in such a way that the series resembles a programming language in its own right. If you have worked with criteria queries in Hibernate ORM, then this style will appear very familiar.

Whether you are using the traditional `FullTextSession` object or the JPA-style `FullTextEntityManager` object, each passes a Lucene query that was generated by the `QueryBuilder` class. This class is the starting point for the Hibernate Search DSL, and it offers several Lucene query types.

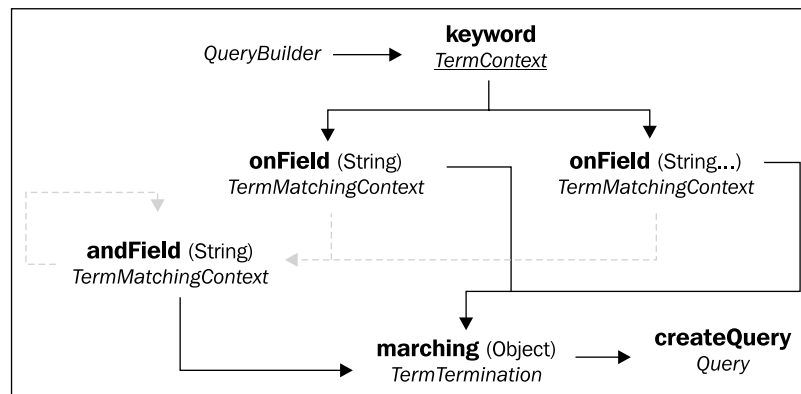
Keyword query

The most basic form of search, which we have glimpsed at already, is the **keyword query**. As the name suggests, this query type searches for one or more particular words.

The first step is to obtain a `QueryBuilder` object, configured for searching on a given entity:

```
...
QueryBuilder queryBuilder =
    fullTextSession.getSearchFactory().buildQueryBuilder()
        .forEntity(App.class).get();
...
```

From there, the following diagram describes the possible flows. Dotted gray arrows represent optional side paths:



Keyword query flow (dotted gray arrows represent optional paths)

In the actual Java code, the DSL for a keyword query would look similar to the following:

```
...
org.apache.lucene.search.Query luceneQuery =
    queryBuilder
        .keyword()
        .onFields("name", "description", "supportedDevices.name",
```

```
        "customerReviews.comments")
        .matching(searchString)
        .createQuery();
    ...
```

The `onField` method takes the name of a field that is indexed for the relevant entity. If the field is not included in that Lucene index, then the query will break. Associated or embedded object fields may also be searched, using the format "[container-field-name].[field-name]" format (for example, `supportedDevices.name`).

Optionally, one or more `andField` methods may be used to search multiple fields. Its parameter works in the exact same way as `onField`. Alternatively, you can declare multiple fields all in one step with `onFields`, as shown in the preceding code snippet.

The `matching` method takes the keyword(s) for which the query is to be searched. This value will generally be a string, although technically the parameter type is a generic object in case you use a field bridge (discussed in the next chapter). Assuming that you pass a string, it may be single keyword or a series of keywords separated by whitespace. By default, Hibernate Search will tokenize the string and search for each keyword individually.

Finally, the `createQuery` method terminates the DSL and returns a Lucene query object. That object may then be used by `FullTextSession` (or `FullTextEntityManager`) to create the final Hibernate Search `FullTextQuery` object:

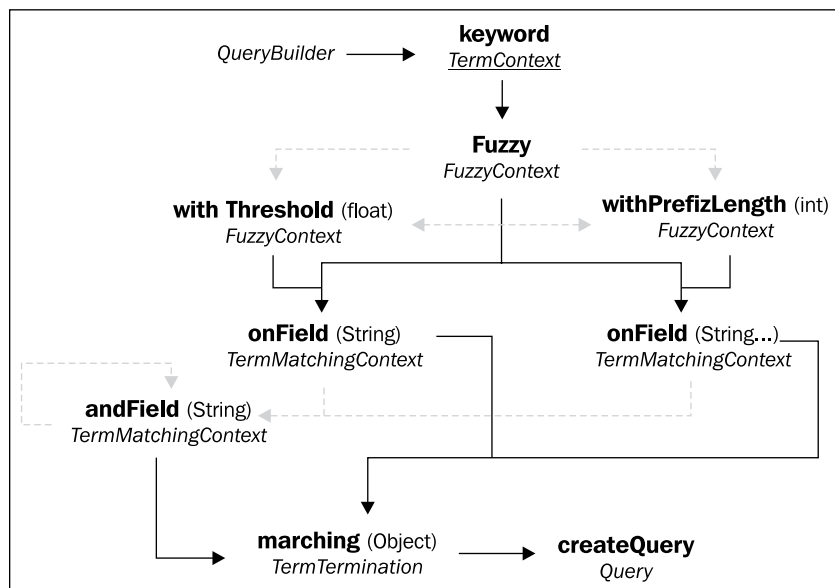
```
    ...
    FullTextQuery hibernateQuery =
        fullTextSession.createFullTextQuery(luceneQuery, App.class);
    ...
```

Fuzzy search

When we use a search engine today, we take for granted that it will be smart enough to fix our typos when we are "close enough" to the correct spelling. One way to add this intelligence to Hibernate Search is by making plain keyword queries **fuzzy**.

With a fuzzy search, keywords match against fields even when they are off by one or more characters. The query runs with a **threshold** value ranging from 0 to 1, where 0 means that everything matches, and 1 means that only exact matches are acceptable. The fuzziness of the query depends on how close to zero you set the threshold.

The DSL starts with the same keyword method and eventually resumes the keyword query flow with `onField` or `onFields`. However, in between are some new flow possibilities, shown as follows:



Fuzzy search flow (dotted gray arrows represent optional paths)

The fuzzy method simply makes a normal keyword query "fuzzy", with a default threshold value of 0.5 (for example, balanced between the two extremes). You can proceed from there with the regular keyword query flow, and that would be perfectly fine.

However, you have the option of calling `withThreshold` to specify a different fuzziness value. In this chapter, versions of the VAPORware Marketplace application add fuzziness to the keyword query, with a threshold value of 0.7. This is strict enough to avoid too many false positives, but fuzzy enough that a misspelled search for "rodio" will now match against the "Athena Internet Radio" app.

```

...
luceneQuery = queryBuilder
    .keyword()
    .fuzzy()
    .withThreshold(0.7f)
    .onFields("name", "description", "supportedDevices.name",
        "customerReviews.comments")
    .matching(searchString)
    .createQuery();
...

```

In addition to (or instead of) `withThreshold`, you may also use `withPrefixLength` to adjust the query fuzziness. This integer value is a number of characters at the beginning of each word that you want to exclude from the fuzziness calculation.

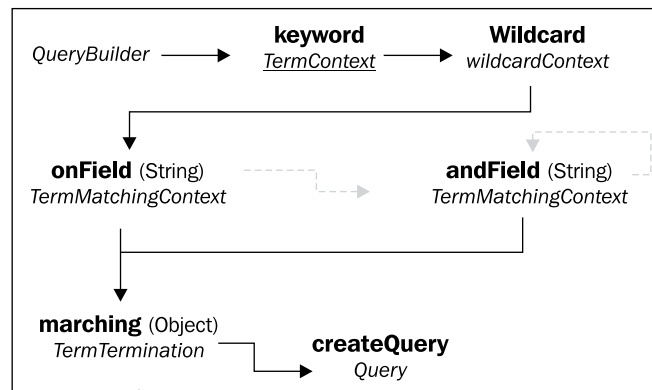
Wildcard search

The second variation on a keyword query doesn't involve any higher math algorithms. If you have ever used a pattern like `*.java` to list all files in a directory, then you already have the basic idea.

Adding the **wildcard** method causes a normal keyword query to treat a question mark (?) as a valid substitute for any single character. For example, the keyword `201?` would match the field values `2010`, `2011`, `2012`, and so on.

The asterisk (*) becomes a substitute for any sequence of zero or more characters. The keyword `down*` matches `download`, `downtown`, and so on.

The Hibernate Search DSL for a wildcard search is the same as that for a regular keyword query, only with the zero-parameter `wildcard` method added at the beginning.



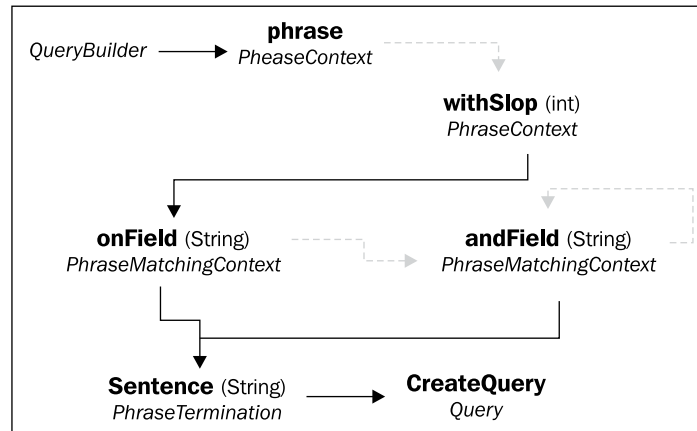
Wildcard search flow (dotted gray arrows represent optional paths)

Exact phrase query

When you type a string of keywords into a search engine, you expect to see results matching one or more of those keywords. Not all of the keywords might be present in each result, and they might not appear in the same order that you typed them.

However, it has become customary that when you place double quotes around a string, you expect the search results to contain that exact phrase.

The Hibernate Search DSL offers a **phrase query** flow for searches of this type.



Exact phrase query flow (dotted gray arrows represent optional paths)

The `onField` and `andField` methods behave in the same way as they do with keyword queries. The `sentence` method differs from `matching` only in that its input must be a `String`.

A primitive form of fuzziness is available to phrase queries, by using the optional `withSlop` clause. This method takes an integer parameter, representing the number of "extra" words that can be found within a phrase before it is no longer considered a match.

This chapter's version of the VAPORware Marketplace application now checks for double quotes around the user's search string. When the input is quoted, the application replaces the keyword query with a phrase query instead:

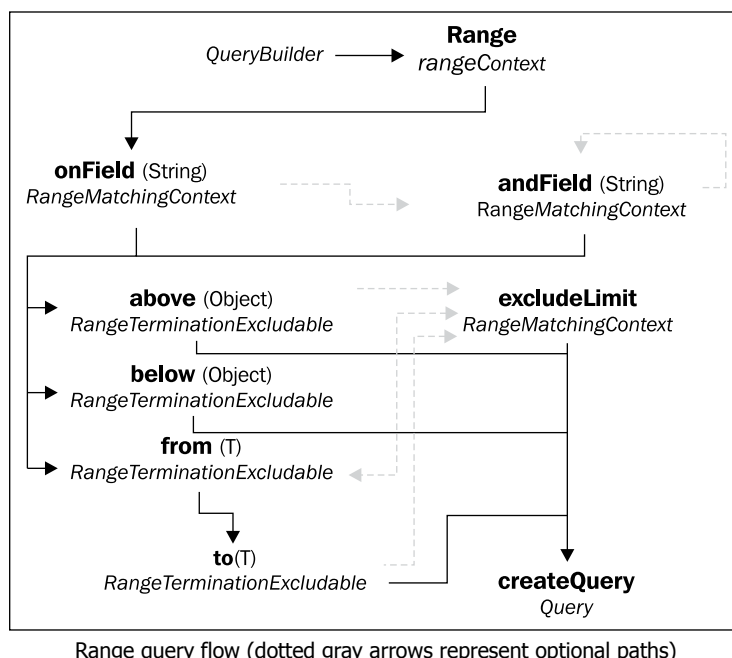
```

...
luceneQuery = queryBuilder
    .phrase()
    .onField("name")
    .andField("description")
    .andField("supportedDevices.name")
    .andField("customerReviews.comments")
    .sentence(searchStringWithQuotesRemoved)
    .createQuery();
...

```

Range query

Phrase queries and the various keyword search types, are all about matching fields to a search term. A **range query** is bit different, in that it looks for fields that are bounded by one or more search terms. In other words, is a field greater than or less than a given value, or in between two values?



When the preceding method is used, the queried field(s) must have values greater than or equal to the input parameter. That parameter is of the generic `Object` type for flexibility. Dates and numeric values are typically used, although strings are perfectly fine and will be compared based on an alphabetical order.

As you might guess, the next method is a counterpart in which values must be less than or equal to the input parameter. To declare that matches must fall in between two parameters, inclusively, you would use the `from` and `to` methods (they must be used together).

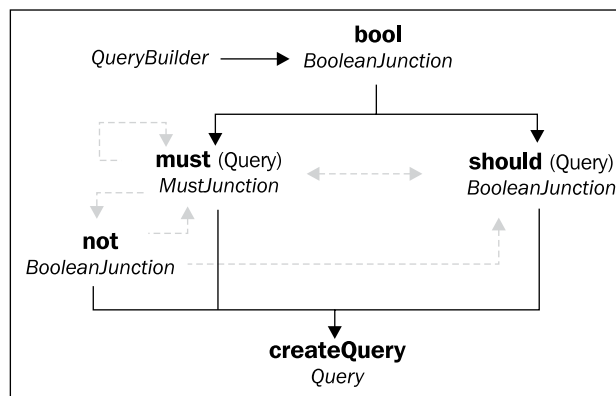
An `excludeLimit` clause may be applied to any of these clauses. It has the effect of making the range exclusive rather than inclusive. In other words, `from(5).to(10).excludeLimit()` matches a range of $5 \leq x < 10$. The modifier could have been placed on the `from` clause rather than the `to`, or on both of them.

In our VAPORware Marketplace application, we previously declined to annotate `CustomerReview.stars` for indexing. However, if we had annotated it with `@Field`, then we could search for all 4- and 5-star reviews with a query similar to the following:

```
...
luceneQuery = queryBuilder
    .range()
    .onField("customerReviews.stars")
    .above(3).excludeLimit()
    .createQuery();
...
```

Boolean (combination) queries

What if you have an advanced use case where a keyword, phrase, or range query is not enough by itself, but two or more of them *together* could meet your requirements? Hibernate Search allows you to mix queries in any combination with boolean logic:



Boolean query flow (dotted gray arrows represent optional paths)

The `bool` method declares that this will be a combination query. It is followed by at least one `must` or `should` clause, each of which takes a Lucene query object of one of the previously discussed varieties.

When a `must` clause is used, a field must match the nested query in order to match the overall query as a whole. Multiple `must` clauses may be applied, which operate in a **logical-AND** fashion. All of them must succeed or else there is no match.

The optional `not` method serves to logically negate a `must` clause. The effect is that the overall query will only match if that nested query doesn't.

The `should` clause roughly approximates a **logical-OR** operation. When a combination consists only of `should` clauses, a field need not match all of them. However, at least one must match in order for the query as a whole to match.



You can combine `must` and `should` clauses. However, if you do so, then the `should` nested queries become completely optional. If the `must` clause succeeds, the overall query succeeds no matter what. If the `must` clause fails, the overall query fails no matter what. When the two clause types are used together, `should` clauses serve only to help rank the search results by relevance.

This example combines a keyword query and a range query to look for "xPhone" apps with 5-star customer reviews:

```
...
luceneQuery = queryBuilder
    .bool()
    .must(
        queryBuilder.keyword().onField("supportedDevices.name")
            .matching("xphone").createQuery()
    )
    .must(
        queryBuilder.range().onField("customerReviews.stars")
            .above(5).createQuery()
    )
    .createQuery();
...
```

Sorting

By default, search results come back in the order of their "relevance". In other words, they are ranked by the degree to which they match the query. We will discuss this further over the next two chapters, and learn how to adjust these relevance calculations.

However, we have the option to change the sorting to some other criteria altogether. In typical situations, you might sort by a date or numeric field, or by a string field in an alphabetical order. In all versions of the VAPORware Marketplace application going forward, users may now sort their search results by the app name.

To sort on a field, special consideration is required when that field is mapped for Lucene indexing. Normally when a string field is indexed, a default analyzer (explored in the next chapter) tokenizes the string. For example, if an `App` entity's name field is "Frustrated Flamingos", then separate entries are created in the Lucene index for "frustrated" and "flamingos". This allows for more powerful querying, but we want to sort based on the original untokenized value.

An easy way to support this is by mapping the field twice, which is perfectly fine! As we saw in *Chapter 2, Mapping Entity Classes*, Hibernate Search offers a plural `@Fields` annotation. It wraps a comma-separated list of `@Field` annotations, with different analyzer settings.

In the following code snippet, one `@Field` is declared with the (tokenizing) defaults. The second one has its `analyze` element sent to `Analyze.NO`, to disable tokenization, and is given its own distinct field name in the Lucene index:

```
...
@Column
@Fields({
    @Field,
    @Field(name="sorting_name", analyze=Analyze.NO)
})
private String name;
...
```

This new field name can be used as follows to build a Lucene `SortField` object, and attach it to a Hibernate Search `FullTextQuery` object:

```
import org.apache.lucene.search.Sort;
import org.apache.lucene.search.SortField;
...
Sort sort = new Sort(
    new SortField("sorting_name", SortField.STRING));
hibernateQuery.setSort(sort); // a FullTextQuery object
```

When a list of search results is later returned by `hibernateQuery`, this list will be sorted by the app name, starting from A to Z.

Reverse sorting is possible as well. The `SortField` class also offers a constructor with a third `Boolean` parameter. If that parameter is set to `true`, the sort will work in the exact opposite manner (for example, Z to A).

Pagination

When a search query returns a huge number of search results, it is usually not desirable (or perhaps even possible) to present them to the user all at once. A common solution is pagination, or displaying search results one "page" at a time.

A Hibernate Search `FullTextQuery` object has methods for making pagination easy:

```
...
hibernateQuery.setFirstResult(10);
hibernateQuery.setMaxResults(5);
List<App> apps = hibernateQuery.list();
...
```

The `setMaxResults` method declares the maximum size of the page. On the last line of the preceding code snippet, the `apps` list will contain no more than five `App` objects, even if the query has thousands of matches.

Of course, pagination wouldn't be very useful if the code always grabbed the first five results. We also need the ability to grab the next page, and then the next page, and so on. So the `setFirstResult` method tells Hibernate Search where to start.

For example, the preceding code snippet starts with the eleventh result item (the parameter is 10, but results are a zero-indexed). The query is then set to grab the next five results. The next incoming request might therefore use `hibernateQuery.setFirstResult(15)`.

The last piece of the puzzle is knowing how many results there are, so you can plan for the correct number of pages:

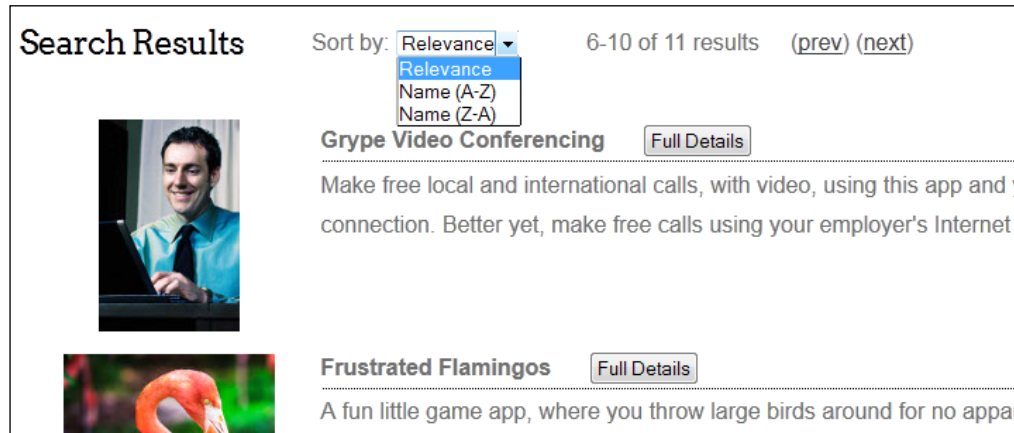
```
...
int resultSize = hibernateQuery.getResultSize();
...
```

The `getResultSize` method is more powerful than it appears at first glance, because it calculates the number using only Lucene indexes. A regular database query across all matching rows could be a very resource-intensive operation, but it is a relatively lightweight matter for Lucene.



This chapter's version of the example applications now use pagination for search results, with a maximum of five results per page. Explore the `SearchServlet` and `search.jsp` results page to see how they use the result size and the current starting point to build the "previous" and "next" links as needed.

A look at the VAPORware Marketplace updates in action is as follows:



Summary

In this chapter, we examined the most common use cases in Hibernate Search querying. We can now work with Hibernate Search regardless of whether JPA is used in whole, in part, or not at all. We learned the core query types offered by the Hibernate Search DSL, and have an easy visual access to all of their possible flows rather than having to crawl through the Javadocs to piece them together.

We now know how to sort search results by a particular field, in ascending or descending order. With large result sets, we can now paginate the results for better performance on the backend, and a better user experience on the frontend. The search functionality in our VAPORware Marketplace example is now greater than or equal to many production Hibernate Search applications.

In the next chapter, we will look at more advanced mapping techniques, such as handling custom data types and controlling details of the Lucene indexing process.

4

Advanced Mapping

So far, we have learned the basics of mapping objects to Lucene indexes. We have seen how to handle relationships with associated entities and embedded objects. However, the searchable fields have mostly been simple string data.

In this chapter, we will look at how to effectively map other data types. We will explore the process by which Lucene analyzes entities for indexing, and the Solr components that can customize that process. We will see how to adjust the importance of each field, to make sorting by relevance more meaningful. Finally, we will conditionally determine whether or not to index an entity at all, based on its state at runtime.

Bridges

The member variables in a Java class may be of an infinite number of custom types. It is usually possible to create custom types in your database as well. With Hibernate ORM, there are dozens of basic types from which more complex types can be constructed.

However, in a Lucene index, everything ultimately boils down to a string. When you map fields of any other data type for searching, the field is converted to a string representation. In Hibernate Search terminology, the code behind this conversion is called a bridge. Default bridges handle most common situations for you transparently, although you have the ability to write your own bridges for custom scenarios.

One-to-one custom conversion

The most common mapping scenario is where a single Java property is tied to a single Lucene index field. The `String` variables obviously don't require any conversion. With most other common data types, how they would be expressed as strings is fairly intuitive.

Mapping date fields

The `Date` values are adjusted to GMT time, and then stored as a string with the format `yyyyMMddHHmmssSSS`.

Although this all happens automatically, you do have the option to explicitly annotate the field with `@DateBridge`. You would do so when you don't want to index down to the exact millisecond. This annotation has one required element, `resolution`, which lets you choose a level of granularity from `YEAR`, `MONTH`, `DAY`, `HOURL`, `MINUTE`, `SECOND`, or `MILLISECOND` (the normal default).

The downloadable `chapter4` version of the VAPORware Marketplace application now adds a `releaseDate` field to the `App` entity. It is configured such that Lucene will only store the day, and not any particular time of day.

```
...
@Column
@Field
@DateBridge(resolution=Resolution.DAY)
private Date releaseDate;
...
```

Handling null values

By default, fields with null values are not indexed regardless of their type. However, you can also customize this behavior. The `@Field` annotation has an optional element, `indexNullAs`, which controls the handling of null values for that mapped field.

```
...
@Column
@Field(indexNullAs=Field.DEFAULT_NULL_TOKEN)
private String description;
...
```

The default setting for this element is `Field.DO_NOT_INDEX_NULL`, which causes null values to be omitted from Lucene indexing. However, when `Field.DEFAULT_NULL_TOKEN` is used, Hibernate Search will index the field with a globally configured value.

The name for this value is `hibernate.search.default_null_token`, and it is set within `hibernate.cfg.xml` (for traditional Hibernate ORM) or `persistence.xml` (for Hibernate configured as a JPA provider). If this value is not configured, then null fields will be indexed with the string `"_null_"`.



You may use this mechanism to apply null-substitution on some fields, and keep the default behavior on other fields. However, the `indexNullAs` element only works with that one substitute value, configured at the global level. If you want to use different null substitutes for different fields or in different scenarios, you must implement that logic through a custom bridge (discussed in the following subsection).

Custom string conversion

Sometimes you need more flexibility in converting a field to a string value. Rather than relying on the built-in bridge to handle it automatically, you can create your own custom bridge.

StringBridge

To map a single Java property to a single index field, your bridge can implement one of two interfaces offered by Hibernate Search. The first of these, `StringBridge`, is for a one-way translation between a property and a string value.

Let's say that our `App` entity has a `currentDiscountPercentage` member variable, representing any promotional discount being offered for that app (for example, *25 percent off!*). For easier math operations, this field is stored as a float (*0.25f*). However, if we ever wanted to make discounts searchable, we would want them indexed in a more human-readable percentage format (*25*).

To provide that mapping, we would start by creating a bridge class, implementing the `StringBridge` interface. The bridge class must implement an `objectToString` method, which expects to take our `currentDiscountPercentage` property as an input parameter:

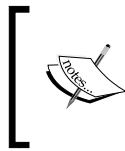
```
import org.hibernate.search.bridge.StringBridge;

/** Converts values from 0-1 into percentages (e.g. 0.25 -> 25) */
public class PercentageBridge implements StringBridge {
```



```
public String objectToString(Object object) {
    try {
        float fieldValue = ((Float) object).floatValue();
        if(fieldValue < 0f || fieldValue > 1f) return "0";
        int percentageValue = (int) (fieldValue * 100);
        return Integer.toString(percentageValue);
    } catch (Exception e) {
        // default to zero for null values or other problems
        return "0";
    }
}
```

The `objectToString` method converts the input as desired, and returns its `String` representation. This will be the value indexed by Lucene.



Notice that this method returns a hardcoded "0" when given a null value, or when it encounters any other sort of problem. Custom null-handling is another possible reason for creating a field bridge.

To invoke this bridge class at index-time, add a `@FieldBridge` annotation to the `currentDiscountPercentage` property:

```
...
@Column
@Field
@FieldBridge(impl=PercentageBridge.class)
private float currentDiscountPercentage;
...
```



This entity field is a primitive float, yet the bridge class is working with a `Float` wrapper object. For flexibility, `objectToString` takes a generic `Object` parameter that must be cast to the appropriate type. However, thanks to autoboxing, primitives are converted into their object wrappers for us seamlessly.

TwoWayStringBridge

The second interface for mapping single variables to single fields, `TwoWayStringBridge`, provides bidirectional translation between a value and its string representation.

You implement `TwoWayStringBridge` in a manner similar to what we just saw with the regular `StringBridge` interface. The only difference is that this bidirectional version also requires a `stringToObject` method, for conversions going the other way:

```
...
public Object stringToObject(String stringValue) {
    return Float.parseFloat(stringValue) / 100;
}
...
```



A bidirectional bridge is only necessary when the field will be an ID field within a Lucene index (that is, annotated with `@Id` or `@DocumentId`).

ParameterizedBridge

For even greater flexibility, it is possible to pass configuration parameters to a bridge class. To do so, your bridge should implement the `ParameterizedBridge` interface, in addition to `StringBridge` or `TwoWayStringBridge`. The class must then implement a `setParameterValues` method for receiving the extra parameters.

For the sake of argument, let's say that we wanted our example bridge to be able to write percentages with a greater level of precision, rather than rounding to a whole number. We could pass it a parameter specifying the number of decimal places to use:

```
public class PercentageBridge implements StringBridge,
    ParameterizedBridge {

    public static final String DECIMAL_PLACES_PROPERTY =
        "decimal_places";
    private int decimalPlaces = 2; // default

    public String objectToString(Object object) {
        String format = "%. " + decimalPlaces + "g%n";
        try {
            float fieldValue = ((Float) object).floatValue();
            if(fieldValue < 0f || fieldValue > 1f) return "0";
            return String.format(format, (fieldValue * 100f));
        } catch (Exception e) {
            return String.format(format, "0");
        }
    }
}
```

```
    public void setParameterValues(Map<String, String> parameters) {
        try {
            this.decimalPlaces = Integer.parseInt(
                parameters.get(DECIMAL_PLACES_PROPERTY) );
        } catch(Exception e) {}
    }
}
```

This version of our bridge class expects to receive a parameter named `decimal_places`. Its value is stored in the `decimalPlaces` member variable, and then used inside the `objectToString` method. If no such parameter is passed, then a default of two decimal places will be used to build percentage strings.

The mechanism for actually passing one or more parameters is the `params` element of the `@FieldBridge` annotation:

```
...
@Column
@Field
@FieldBridge(
    impl=PercentageBridge.class,
    params=@Parameter(
        name=PercentageBridge.DECIMAL_PLACES_PROPERTY, value="4")
)
private float currentDiscountPercentage;
...
```



Be aware that all implementations of `StringBridge` or `TwoWayStringBridge` must be thread-safe. Generally, you should avoid any shared resources, and only take additional information through the `ParameterizedBridge` parameters.

More complex mappings with FieldBridge

The bridge types covered so far are the easiest and most straightforward way to map a Java property to a string index value. However, sometimes you need even greater flexibility, so there are a few field bridge variations supporting a free-form approach.

Splitting a single variable into multiple fields

Occasionally, the desired relationship between a class property and Lucene index fields may not be one-to-one. For example, let's say that one property represents a filename. However, we would like the ability to search not only by filename, but also by file type (that is, the file extension). One approach is to parse the file extension from the filename property, and thereby use that one variable to create both fields.

The `FieldBridge` interface allows us to do this. Implementations must provide a `set` method, which in this example parses the file type from the file name field, and stores them separately:

```
import org.apache.lucene.document.Document;
import org.hibernate.search.bridge.FieldBridge;
import org.hibernate.search.bridge.LuceneOptions;

public class FileBridge implements FieldBridge {

    public void set(String name, Object value,
        Document document, LuceneOptions luceneOptions) {
        String file = ((String) value).toLowerCase();
        String type = file.substring(
            file.indexOf(".") + 1 ).toLowerCase();
        luceneOptions.addFieldToDocument(name+".file", file, document);
        luceneOptions.addFieldToDocument(name+".file_type", type,
            document);
    }
}
```

The `luceneOptions` parameter is a helper object for interacting with Lucene, and `document` represents the Lucene data structure to which we are adding fields. We use `luceneOptions.addFieldToDocument()` to add fields to the index, without having to fully understand the Lucene API details.

The `name` parameter passed to `set` represents the name of the entity being indexed. Notice that we use this as a base when declaring the names of the two entities being added (that is, `name+".file"` for the filename, and `name+".file_type"` for the file type).

Finally, the `value` parameter is the current field being mapped. Just as with the `StringBridge` interface seen in the `Bridges` section, the method signature here uses a generic `Object` for flexibility. The value must be cast to its appropriate type.

To apply a `FieldBridge` implementation, use the `@FieldBridge` annotation just as we've already seen with the other custom bridge types:

```
...
@Column
@Field
@FieldBridge(impl=FileBridge.class)
private String file;
...
```

Combining multiple properties into a single field

A custom bridge implementing the `FieldBridge` interface may also be used for the reverse purpose, to combine more than one property into a single index field. To gain this degree of flexibility, the bridge must be applied to the *class* level rather than the *field* level. When the `FieldBridge` interface is used in this manner, it is known as a **class bridge**, and replaces the usual mapping mechanism for the entire entity class.

For example, consider an alternate approach we could have taken with the `Device` entity in our VAPORware Marketplace application. Instead of indexing `manufacturer` and `name` as separate fields, we could have combined them into one `fullName` field. The class bridge for this would still implement the `FieldBridge` interface, but it would concatenate the two properties into one index field as follows:

```
public class DeviceClassBridge implements FieldBridge {

    public void set(String name, Object value,
        Document document, LuceneOptions luceneOptions) {
        Device device = (Device) value;
        String fullName = device.getManufacturer()
            + " " + device.getName();
        luceneOptions.addFieldToDocument(name + ".name",
            fullName, document);
    }
}
```

Rather than applying an annotation to any particular fields within the `Device` class, we would instead apply a `@ClassBridge` annotation at the class level. Notice that the field-level Hibernate Search annotations have been completely removed, as the class bridge will be responsible for mapping all index fields in this class.

```
@Entity
@Indexed
@ClassBridge(impl=DeviceClassBridge.class)
public class Device {

    @Id
    @GeneratedValue
    private Long id;

    @Column
    private String manufacturer;

    @Column
    private String name;

    // constructors, getters and setters...
}
```

TwoWayFieldBridge

Earlier we saw that the simple `StringBridge` interface has a `TwoWayStringBridge` counterpart, providing bidirectional mapping capability for document ID fields. Likewise, the `FieldBridge` interface has a `TwoWayFieldBridge` counterpart for the same reason. When you apply a field bridge interface to a property used by Lucene as an ID (that is, annotated with `@Id` or `@DocumentId`), then you must use the two-way variant.

The `TwoWayStringBridge` interface requires the same `objectToString` method as `StringBridge`, and the same `set` method as `FieldBridge`. However, this two-way version also requires a `get` counterpart, for retrieving the string representation from Lucene and converting if the true type is different:

```
...
public Object get(String name, Object value, Document document) {
    // return the full file name field... the file type field
    // is not needed when going back in the reverse direction
    return = document.get(name + ".file");
}
```

```
public String objectToString(Object object) {  
    // "file" is already a String, otherwise it would need conversion  
    return object;  
}  
...
```

Analysis

When a field is indexed by Lucene, it undergoes a parsing and conversion process called **analysis**. In *Chapter 3, Performing Queries*, we mentioned that the default **analyzer** tokenizes string fields, and that this behavior should be disabled if you plan to sort on that field.

However, much more is possible during analysis. Apache Solr components may be assembled in hundreds of combinations. They can manipulate text in various ways during indexing, and open the door to some really powerful search functionally.

In order to discuss the Solr components that are available, or how to assemble them into a custom analyzer definition, we must first understand the three phases of Lucene analysis:

- Character filtering
- Tokenization
- Token filtering

Analysis begins by applying zero or more **character filters**, which strip or replace characters prior to any other processing. The filtered string then undergoes **tokenization**, splitting it into smaller tokens to make keyword searches more efficient. Finally, zero or more **token filters** remove or replace tokens before they are saved to the index.



These components are provided by the Apache Solr project, and they number over three-dozen in total. This book cannot dive deeply into every single one, but we can take a look at a few key examples of the three types and see how to apply them generally. The full documentation for all of these Solr analyzer components may be found at <http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters>, with Javadocs at http://lucene.apache.org/solr/api-3_6_1.

Character filtering

When defining a custom analyzer, character filtering is an optional step. Should this step be desired, there are only three character filter types available:

- `MappingCharFilterFactory`: This filter replaces characters (or sequences of characters) with specifically defined replacement text, for example, you might replace occurrences of *1* with *one*, *2* with *two*, and so on.
The mappings between character(s) and replacement value(s) are stored in a resource file, using the standard `java.util.Properties` format, located somewhere in the application's classpath. For each property, the key is the sequence to look for, and the value is the mapped replacement.
The classpath-relative location of this mappings file is passed to the `MappingCharFilterFactory` definition, as a parameter named `mapping`. The exact mechanism for passing this parameter will be illustrated shortly in the *Defining and Selecting Analyzers* section.
- `PatternReplaceCharFilter`: This filter applies a regular expression, passed via a parameter named `pattern`. Any matches will be replaced with a string of static text passed via a `replacement` parameter.
- `HTMLStripCharFilterFactory`: This extremely useful filter removes HTML tags, and replaces escape sequences with their usual text forms (for example, `>` becomes `>`).

Tokenization

Character and token filters are both optional when defining a custom analyzer, and you may combine multiple filters of both types. However, the tokenizer component is unique. An analyzer definition must contain one, and no more than one.

There are 10 tokenizer components available in total. Some illustrative examples include:

- `WhitespaceTokenizerFactory`: This simply splits text on whitespace. For instance, *hello world* is tokenized into *hello* and *world*.
- `LetterTokenizerFactory`: This is similar to `WhitespaceTokenizerFactory` in functionality, but this tokenizer also splits text on non-letter characters. The non-letter characters are discarded altogether, for example, *please don't go* is tokenized into *please*, *don*, *t*, and *go*.
- `StandardTokenizerFactory`: This is the default tokenizer that is automatically applied when you don't define a custom analyzer. It generally splits on whitespace, discarding extraneous characters. For instance, *it's 25.5 degrees outside!!!* becomes *it's*, *25.5*, *degrees*, and *outside*.



When in doubt, `StandardTokenizerFactory` is almost always the sensible choice.

Token filtering

By far the greatest variety in analyzer functionality comes through token filters, with Solr offering two dozen options for use alone or in combination. These are only a few of the more useful examples:

- `StopFilterFactory`: This filter simply throws away "stop words", or extremely common words for which no one would ever want to perform a keyword query anyway. The list includes *a, the, if, for, and, or*, and so on (the Solr documentation presents the full list).
- `PhoneticFilterFactory`: When you use a major search engine, you would probably notice that it can be very intelligent in dealing with your typos. One technique for doing this is to look for words that sound similar to the searched keyword, in case it was misspelled. For example, if you meant to search for *morning*, but misspelled it as *mourning*, the search would still match the intended term! This token filter provides that functionality by indexing phonetically similar strings along with the actual token. The filter requires a parameter named `encoder`, set to the name of a supported encoding algorithm ("DoubleMetaphone" is a sensible option).
- `SnowballPorterFilterFactory`: Stemming is a process in which tokens are broken down into their root form, to make it easier to match related words. Snowball and Porter refer to stemming algorithms. For instance, the words *developer* and *development* can both be broken down to the root stem *develop*. Therefore, Lucene can recognize a relationship between the two longer words (even though neither one is a substring of the other!) and can return matches on both. This filter takes one parameter, named `language` (for example, "English").

Defining and selecting analyzers

An **analyzer definition** assembles some combination of these components into a logical whole, which can then be referenced when indexing an entity or individual field. Custom analyzers may be defined in a static manner, or may be assembled dynamically based on some conditions at runtime.

Static analyzer selection

Either approach for defining a custom analyzer begins with an `@AnalyzerDef` annotation on the relevant persistent class. In the `chapter4` version of our VAPORware Marketplace application, let's define a custom analyzer to be used with the `App` entity's `description` field. It should strip out any HTML tags, and apply various token filters to reduce clutter and account for typos:

```
...
@AnalyzerDef(
    name="appAnalyzer",
    charFilters={
        @CharFilterDef(factory=HTMLStripCharFilterFactory.class)
    },
    tokenizer=@TokenizerDef(factory=StandardTokenizerFactory.class),
    filters={
        @TokenFilterDef(factory=StandardFilterFactory.class),
        @TokenFilterDef(factory=StopFilterFactory.class),
        @TokenFilterDef(factory=PhoneticFilterFactory.class,
            params = {
                @Parameter(name="encoder", value="DoubleMetaphone")
            }),
        @TokenFilterDef(factory=SnowballPorterFilterFactory.class,
            params = {
                @Parameter(name="language", value="English")
            })
    }
)
...
```

The `@AnalyzerDef` annotation must have a `name` element set, and as previously discussed, an analyzer must always include one and only one tokenizer.

The `charFilters` and `filters` elements are optional. If set, they receive lists of one or more factory classes, for character filters and token filters respectively.



Be aware that character filters and token filters are applied *in the order they are listed*. In some cases, changes to the ordering can affect the final result.

The `@Analyzer` annotation is used to select and apply a custom analyzer. This annotation may be placed on an individual field, or on the overall class where it will affect every field. In this case, we are only selecting our analyzer definition for the `description` field:

```
...
@Column(length = 1000)
@Field
@Analyzer(definition="appAnalyzer")
private String description;
...
```

It is possible to define multiple analyzers in a single class, by wrapping their `@AnalyzerDef` annotations within a plural `@AnalyzerDefs`:

```
...
@AnalyzerDefs({
    @AnalyzerDef(name="stripHTMLAnalyzer", ...),
    @AnalyzerDef(name="applyRegexAnalyzer", ...)
})
...
```

Obviously, where the `@Analyzer` annotation is later applied, its definition element has to match the appropriate `@AnalyzerDef` annotation's name element.



The chapter4 version of the VAPORware Marketplace application now strips HTML from the customer reviews. If a search includes the keyword *span*, there will not be a false positive match on reviews containing the `` tag, for instance. Snowball and phonetic filters are being applied to the app descriptions. The keyword *mourning* finds a match containing the word *morning*, and a search for *development* returns an app with *developers* in its description.

Dynamic analyzer selection

It is possible to wait until runtime to select a particular analyzer for a field. The most obvious scenario is an application supporting different languages, with analyzer definitions configured for each language. You would want to select the appropriate analyzer based on a language attribute for each object.

To support such a dynamic selection, an `@AnalyzerDiscriminator` annotation is added to a particular field or to the class as a whole. This code snippet uses the latter approach:

```
@AnalyzerDefs({
    @AnalyzerDef(name="englishAnalyzer", ...),
    @AnalyzerDef(name="frenchAnalyzer", ...)
})
@AnalyzerDiscriminator(impl=CustomerReviewDiscriminator.class)
public class CustomerReview {
    ...
    @Field
    private String language;
    ...
}
```

There are two analyzer definitions, one for English and the other for French, and the class `CustomerReviewDiscriminator` is declared responsible for deciding which to use. This class must implement the `Discriminator` interface, and its `getAnalyzerDefinitionName` method:

```
public class LanguageDiscriminator implements Discriminator {

    public String getAnalyzerDefinitionName(Object value,
        Object entity, String field) {
        if( entity == null || !(entity instanceof CustomerReview) ) {
            return null;
        }
        CustomerReview review = (CustomerReview) entity;
        if(review.getLanguage() == null) {
            return null;
        } else if(review.getLanguage().equals("en")) {
            return "englishAnalyzer";
        } else if(review.getLanguage().equals("fr")) {
            return "frenchAnalyzer";
        } else {
            return null;
        }
    }
}
```

If the `@AnalyzerDiscriminator` annotation is placed on a field, then its value for the current object is automatically passed as the first parameter to `getAnalyzerDefinitionName`. If the annotation is placed on the class itself, then a null value is passed instead. The second parameter is the current entity object either way.

In this case, the discriminator is applied at the class level. So we cast that second parameter to type `CustomerReview`, and return the name of the appropriate analyzer based on the object's `language` field. If the language is unknown or if there are other issues, then the method simply returns `null`, telling Hibernate Search to fall back to the default analyzer.

Boosting search result relevance

We have already seen that the default sort order for search results is by relevance, meaning the degree to which they match the query. If one entity matches on two fields, while another has only one field match, then that first entity is the more relevant result.

Hibernate Search allows us to adjust how relevance is calculated, by **boosting** the relative importance of entities or fields when they are indexed. These adjustments can be static and fixed, or they can be dynamic and driven by the state of the data at runtime.

Static boosting at index-time

Fixed boosting, regardless of the actual data, is as simple as annotating a class or field with `@Boost`. This annotation takes a floating-point parameter for its relative weight, with the default weight being 1.0. So for example, `@Boost(2.0f)` would double the weight of a class or field relative to non-annotated classes and fields.

Our VAPORware Marketplace application searches on several fields and associations, such as the names of supported devices, and comments posted in customer reviews. However, doesn't it make sense that the text under our control (each app's name and full description) should carry more weight than text coming from outside parties?

To make this adjustment, the `chapter4` version starts by annotating the `App` class itself:

```
...
@Boost(2.0f)
public class App implements Serializable {
...

```

This essentially makes App twice as relevant as Device or CustomerReview. Next, we apply field-level boosting to the name and full description fields:

```
...
@Boost(1.5f)
private String name;
...
@Boost(1.2f)
private String description;
...
```

We are declaring here that name carries slightly more weight than description, and they each carry more weight relative to normal fields.



Be aware that class-level and field-level boosting cascade and combine! When more than one boost factor applies to a given field, they are multiplied to form the total factor. Here, because a weight of 2.0 was already applied to the App class itself, name has a total effective weight of 3.0 and description is at 2.4.

Dynamic boosting at index-time

For an example of boosting an entity dynamically based on its data at index-time, let's say that we wanted to give the CustomerReview objects a bit more weight when the reviewer gives a five-star rating. To do this, we apply a @DynamicBoost annotation to the class:

```
...
@DynamicBoost(impl=FiveStarBoostStrategy.class)
public class CustomerReview {
...

```

This annotation must be passed a class that implements the BoostStrategy interface, and its defineBoost method:

```
public class FiveStarBoostStrategy implements BoostStrategy {

    public float defineBoost(Object value) {
        if(value == null || !(value instanceof CustomerReview)) {
            return 1;
        }
        CustomerReview customerReview = (CustomerReview) value;
        if(customerReview.getStars() == 5) {
```

```
        return 1.5f;
    } else {
        return 1;
    }
}
```

When the `@DynamicBoost` annotation was applied to a class, the parameter automatically passed to `defineBoost` is an instance of that class (a `CustomerReview` object in this case). If the annotation had been applied to a particular field, then the automatically-passed parameter would be that field's value.

The `float` value returned by `defineBoost` becomes the weight of the class or field that was annotated. In this case, we increase a `CustomerReview` object's weight to 1.5 when it represents a five-star review. Otherwise, we keep the 1.0 default.

Conditional indexing

There are specialized ways to go about indexing fields, such as using a class bridge or the programmatic mapping API. Generally speaking, though, a property is indexed when it is annotated with `@Field`. Therefore, one obvious way to avoid indexing a field is to simply not apply the annotation.

However, what if we want an entity class to be searchable in general, but we need to exclude certain instances of that class, based on the state of their data at runtime?

The `@Indexed` annotation has an experimental second element, `interceptor`, that gives us the ability to index conditionally. When this element is set, the normal indexing process will be intercepted by custom code, which can prevent an entity from being indexed based on its current state.

Let's give our VAPORware Marketplace the ability to make apps inactive. Inactive apps will still exist in the database, but should not be shown to customers or indexed for searching. To start, we will add a new property to the `App` entity class:

```
...
@Column
private boolean active;
...
public App(String name, String image, String description) {
    this.name = name;
    this.image = image;
    this.description = description;
    this.active = true;
}
```

```

    }
    ...
    public boolean isActive() {
        return active;
    }
    public void setActive(boolean active) {
        this.active = active;
    }
    ...

```

This new active variable has the standard getter and setter methods, and is being defaulted to true in our normal constructor. We want individual apps to be excluded from the Lucene index when this variable is false, so we add an interceptor element to the @Indexed annotation:

```

...
import com.packtpub.hibernatesearch.util.IndexWhenActiveInterceptor;
...
@Entity
@Indexed(interceptor=IndexWhenActiveInterceptor.class)
public class App {
    ...

```

This element must be tied to a class that implements the EntityIndexingInterceptor interface. Since we just specified a class named IndexWhenActiveInterceptor, we need to now create this class.

```

package com.packtpub.hibernatesearch.util;

import org.hibernate.search.indexes.interceptor.
    EntityIndexingInterceptor;
import org.hibernate.search.indexes.interceptor.IndexingOverride;
import com.packtpub.hibernatesearch.domain.App;

public class IndexWhenActiveInterceptor
    implements EntityIndexingInterceptor<App> {

    /** Only index newly-created App's when they are active */
    public IndexingOverride onAdd(App entity) {
        if(entity.isActive()) {
            return IndexingOverride.APPLY_DEFAULT;
        }
        return IndexingOverride.SKIP;
    }
}

```



```
public IndexingOverrideonDelete(App entity) {
    return IndexingOverride.APPLY_DEFAULT;
}

/** Index active App's, and remove inactive ones */
public IndexingOverrideonUpdate(App entity) {
    if(entity.isActive()) {
        return IndexingOverride.UPDATE;
    } else {
        return IndexingOverride.REMOVE;
    }
}

public IndexingOverrideonCollectionUpdate(App entity) {
    return nonUpdate(entity);
}
}
```

The `EntityIndexingInterceptor` interface declares **four methods**, which Hibernate Search will call at various points during an entity object's life cycle:

- `onAdd()`: This is called when the entity instance is first created.
- `onDelete()`: This is called when the entity instance is removed from the database.
- `onUpdate()`: This is called when an existing instance is updated.
- `onCollectionUpdate()`: This version is used when an entity is modified as part of a bulk update with other entities. Typically, implementations of this method simply invoke `onUpdate()`.

Each of these methods should return one of the four possible `IndexingOverride` enum values. The possible **return values** tell Hibernate Search what to do:

- `IndexingOverride.SKIP`: This tells Hibernate Search to not modify the Lucene index for this entity instance at this time.
- `IndexingOverride.REMOVE`: Hibernate Search will remove the entity if it already exists in an index, or else will do nothing if the entity is not indexed.
- `IndexingOverride.UPDATE`: The entity will be updated in the index, or added if it is not already indexed.

- `IndexingOverride.APPLY_DEFAULT`: This is equivalent to the custom interceptor not being used in the first place. Hibernate Search will index the entity if this is an `onAdd()` operation, remove it from the index if this is an `onDelete()`, or update the index if this is `onUpdate()` or `onCollectionUpdate()`.

Although the four methods logically imply certain return values, it is actually possible to mix them in any combination if you are dealing with unusual conditions.

In our example application, our interceptor examines the entity in `onAdd()` and `onDelete()`. When a new `App` is created, indexing is skipped if its `active` variable is `false`. When an `App` is updated, it will be removed from the index if has become inactive.

Summary

In this chapter, we toured the full range of functionality available in mapping persistent objects for search. We can now adjust settings for Hibernate Search's built-in type bridges, and can create highly advanced custom bridges of our own.

We now have a deeper understanding of Lucene analysis. We have worked with some of the most useful custom analyzer components, and know how to independently obtain information on dozens of other Solr components.

We are now able to adjust the relative weight of classes and fields through boosting, to improve the quality of our search results when there are sorted by relevance. Last but not least, we learned how to use conditional indexing to dynamically prevent certain data from being searchable based on its state.

In the next chapter, we will turn to more advanced query concepts. We will learn how to filter and categorize search results, and pull data from Lucene alone without needing a database call.

5

Advanced Querying

In this chapter, we will elaborate on the basic search query concepts that we covered earlier, in light of the new mapping knowledge that we just picked up. We will now look at a number of techniques for making search queries more flexible and powerful.

We will see how to dynamically filter results at the Lucene level, before the database is even touched. We will also avoid database calls by using projection-based queries, to retrieve properties directly from Lucene. We will use faceted search, to recognize and isolate subsets of data within search results. Finally, we will cover some miscellaneous query tools, such as query-time boosting and placing time limits on a query.

Filtering

The process of building a query revolves around finding matches. However, sometimes you want to narrow the search results on the basis of a criteria that explicitly did *not* match. For example, let's say we want to limit our VAPORware Marketplace search to only those apps that are supported on a particular device:

- Adding a keyword or phrase to an existing query doesn't help, because that would just make the query more inclusive.
- We could turn the existing query into a boolean query, with an extra `must` clause, but then the DSL starts to become harder to maintain. Also, if you need to use complex logic to narrow your results, then the DSL may not offer enough flexibility.
- A Hibernate Search `FullTextQuery` object inherits from the Hibernate ORM `Query` (or its JPA counterpart) class. So, we can narrow results using core Hibernate tools like `ResultTransformer`. However, this requires making additional database calls, which can impact performance.

Hibernate Search offers a more elegant and efficient **filter** approach. Through this mechanism, filter logic for various scenarios is encapsulated in separate classes. Those filter classes may be dynamically enabled or disabled at runtime, alone or in any combination. When a query is filtered, unwanted results are never fetched from Lucene in the first place. This reduces the weight of any follow-up database access.

Creating a filter factory

To filter our search results by supported devices, the first step is creating a class to store the filtering logic. This should be an instance of `org.apache.lucene.search.Filter`. For simple hardcoded logic, you might just create your own subclass.

However, if we instead generate filters dynamically with a filter factory, then we can accept parameters (for example, device name) and customize the filter at runtime:

```
public class DeviceFilterFactory {

    private String deviceName;

    @Factory
    public Filter getFilter() {
        PhraseQuery query = new PhraseQuery();
        StringTokenizer tokenzier = new StringTokenizer(deviceName);
        while(tokenzier.hasMoreTokens()) {
            Term term = new Term(
                "supportedDevices.name", tokenzier.nextToken());
            query.add(term);
        }
        Filter filter = new QueryWrapperFilter(query);
        return new CachingWrapperFilter(filter);
    }

    public void setDeviceName(String deviceName) {
        this.deviceName = deviceName.toLowerCase();
    }

}
```

The `@Factory` annotation is applied to the method responsible for producing the Lucene filter object. In this case, we annotate the aptly named `getFilter` method.



Unfortunately, building a `Lucene Filter` object requires us to work more closely with the raw Lucene API, rather the convenient DSL wrapper provided by Hibernate Search. The full Lucene API is very involved, and covering it completely would require an entirely separate book. However, even this shallow dive is deep enough to give us the tools for writing really useful filters.

This example builds a filter by wrapping a Lucene query, and then applying a second wrapper to facilitate filter caching. A specific type of query used is `org.apache.lucene.search PhraseQuery`, which is equivalent to the DSL phrase query that we explored in *Chapter 3, Performing Queries*.



We are examining the phrase query in this example, because it is one of the most useful types for a building a filter. However, there are 15 Lucene query types in total. You can explore the JavaDocs at http://lucene.apache.org/core/old_versioned_docs/versions/3_0_3/api/all/org/apache/lucene/search/Query.html.

Let's review some of the things we know about how data is stored in a Lucene index. By default, an analyzer tokenizes strings, and indexes them as individual terms. The default analyzer also converts the string data into lowercase. The Hibernate Search DSL normally hides all of this detail, so developers don't have to think about it.

However, you do need to account for these things when using the Lucene API directly. Therefore, our `setDeviceName` setter method manually converts the `deviceName` property to lower case, to avoid a mismatch with Lucene. The `getFilter` method then manually tokenizes this property into separate terms, likewise to match what Lucene has indexed.

Each tokenized term is used to construct a Lucene `Term` object, which consists of the data and the relevant field name (that is, `supportedDevices.name` in this case). These terms are added to the `PhraseQuery` object one by one, in the exact order that they appear in the phrase. The query object is then wrapped up as a filter and returned.

Adding a filter key

By default, Hibernate Search caches filter instances for better performance. Therefore, each instance requires that a unique key be referenced by in the cache. In this example, the most logical key would be the device name for which each instance is filtering.

First, we add a new method to our filter factory, annotated with `@Key` to indicate that it is responsible for generating the unique key. This method returns a subclass of `FilterKey`:

```
...
@Key
public FilterKey getKey() {
    DeviceFilterKey key = new DeviceFilterKey();
    key.setDeviceName(this.deviceName);
    return key;
}
...
```

Custom `FilterKey` subclasses must implement the methods `equals` and `hashCode`. Typically, when the actual wrapped data may be expressed as a string, you can delegate to the corresponding methods on the `String` class:

```
public class DeviceFilterKey extends FilterKey {

    private String deviceName;

    @Override
    public boolean equals(Object otherKey) {
        if(this.deviceName == null
            || !(otherKey instanceof DeviceFilterKey)) {
            return false;
        }
        DeviceFilterKey otherDeviceFilterKey =
            (DeviceFilterKey) otherKey;
        return otherDeviceFilterKey.deviceName != null
            && this.deviceName.equals(otherDeviceFilterKey.
deviceName);
    }

    @Override
    public int hashCode() {
        if(this.deviceName == null) {
            return 0;
        }
        return this.deviceName.hashCode();
    }

    // GETTER AND SETTER FOR deviceName...
}
```

Establishing a filter definition

To make this filter available for our app searches, we will create a filter definition in the App entity class:

```
...
@FullTextFilterDefs({
    @FullTextFilterDef(
        name="deviceName", impl=DeviceFilterFactory.class
    )
})
public class App {
    ...
}
```

The `@FullTextFilterDef` annotation links the entity class with a given filter or filter-factory class, specified by the `impl` element. The `name` element is a string by which Hibernate Search queries can reference the filter, as we'll see in the next subsection.

An entity class may have any number of defined filters. The plural `@FullTextFilterDefs` annotation supports this, by wrapping a comma-separated list of one or more singular `@FullTextFilterDef` annotations.

Enabling the filter for a query

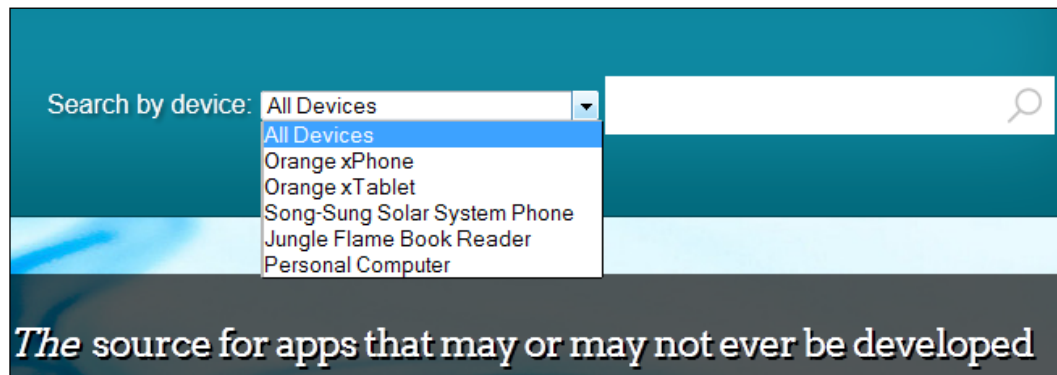
Last but not least, we enable the filter definition for a Hibernate Search query, using the `FullTextQuery` object's `enableFullTextFilter` method:

```
...
if(selectedDevice != null && !selectedDevice.equals("all")) {
    hibernateQuery.enableFullTextFilter("deviceName")
        .setParameter("deviceName", selectedDevice);
}
...
```

This method's `string` parameter is matched to a filter definition on one of the entity classes involved in the query. In this case, it's the `deviceName` filter defined on `App`. When Hibernate Search finds this match, it will automatically invoke the corresponding filter factory to get a `Filter` object.

Our filter factory uses a parameter, also called `deviceName` for consistency (although it's a different variable). Before Hibernate Search can invoke the factory method, this parameter must be set, by passing the parameter name and value to `setParameter`.

The filter is enabled within an `if` block, so that we can skip this when no device was selected (that is, the **All Devices** option). If you examine the downloadable code bundle for this chapter's version of the VAPORware Marketplace application, you will see that the HTML file has been modified to add a drop-down menu for device selection:



Projection

In the first couple of chapters, our example application fetched all the matching entities in one big database call. We introduced pagination in *Chapter 3, Performing Queries*, to at least limit the database calls to a fixed number of rows. However, since we're already searching data in a Lucene index to begin with, is it really necessary to go to the database at all?

Hibernate Search offers **projections** as a technique for eliminating, or at least reducing, database access. A projection-based search returns only specific fields pulled from Lucene, rather than returning a full entity object from the database. You can then go to the database and fetch full objects if necessary, but the fields available in Lucene may be sufficient by themselves.

This chapter's version of the VAPORware Marketplace application modifies the search results page so that it now uses a projection-based query. The previous versions of the page received App entities all at once, and hid each app's pop-up window until its **Full Detail** button was clicked. Now, the page receives only enough fields to build the summary view. Each **Full Detail** button triggers an AJAX call for that app. Only then is the database called, and only to fetch data for that one app.



Exhaustive descriptions of how to make AJAX calls from JavaScript and how to write RESTful web services to respond to those calls, ventures pretty far beyond the scope of this Hibernate Search book.

That being said, all of the JavaScript is contained on the search results JSP, within the `showAppDetails` function. All of the corresponding server-side Java code resides in the `com.packtpub.hibernatesearch.rest` package, and is heavily commented. There are endless online primers and tutorials for writing RESTful services, and the documentation for the particular framework used here is at <http://jersey.java.net/nonav/documentation/latest>.

Making a query projection-based

To change `FullTextQuery` to be projection-based, invoke the `setProjection` method on that object. Our search servlet class now contains the following line:

```
...
hibernateQuery.setProjection("id", "name", "description", "image");
...
```

The method accepts the names of one or more fields to pull from the Lucene indexes associated with this query.

Converting projection results to an object form

If we stopped right here, then the query object's `list()` method would no longer return a list of `App` objects! By default, projection-based queries return a list of object arrays (that is, `Object[]`) instead of entity objects. These arrays are often referred to as **tuples**.

The elements in each tuple contain values for the projected fields, in the order they were declared. For example, here `listItem[0]` would contain the value of a result's ID, `field.listItem[1]` would contain the name, `value.listItem[2]` would contain the description, and so on.

In some cases, it's easy enough to work with the tuple as-is. However, you can automatically convert tuples into an object form by attaching a Hibernate ORM result transformer to the query. Doing so changes the query's return type yet again, from `List<Object[]>` to a list of the desired object type:

```
...
hibernateQuery.setResultTransformer(
    newAliasToBeanResultTransformer(App.class) );
...
```

You can create your own custom transformer class inheriting from `ResultTransformer`, implementing whatever complex logic you need. However, in most cases, the subclasses provided by Hibernate ORM out of the box are more than enough.

Here, we are using the `AliasToBeanResultTransformer` subclass, and initializing it with our `App` entity class. This matches up the projected fields with the entity class properties having the same names, and sets each property with the corresponding field value.

Only a subset of properties of `App` are available. It is okay to leave the other properties uninitialized, since the search results JSP doesn't need them when building its summary list. Also, the resulting `App` objects won't actually be attached to a Hibernate session. However, we've been detaching our results before sending them to the JSP anyway.

Making Lucene fields available for projection

By default, Lucene indexes are optimized with the assumption that they will not be used for projection-based queries. Therefore, projection requires that you make some small mapping changes and bear a couple of caveats in mind.

First and foremost, the field data must be stored by Lucene in a manner that can be easily retrieved. The normal indexing process optimizes data for complex queries, not for retrieval in its original form. To store a field's value in a form that can be restored by a projection, you add a `store` element to the `@Field` annotation:

```
...
@Field(store=Store.COMPRESS)
private String description;
...
```

This element takes an enum with three possible values:

- `Store.NO` is the default. It causes the field to be indexed for searching, but not retrievable in its original form through projection.
- `Store.YES` causes the field to be included as-is in the Lucene index. This increases the size of the index, but makes projections possible.
- `Store.COMPRESS` is an attempt at compromise. It also stores the field as-is, but applies compression to reduce the overall index size. Be aware that this is more processor-intensive, and is not available for a field that also uses the `@NumericField` annotation.

Secondly, a field must use a bi-directional field bridge. All of the default bridges built-in to Hibernate Search already support this. However, if you create your own custom bridge type (see *Chapter 4, Advanced Mapping*), it must be based on `TwoWayStringBridge` or `TwoWayFieldBridge`.

Last but not least, projection is only effective for basic properties on the entity class itself. It is not meant for fetching associated entities or embedded objects. If you do try to reference an association, then you will only get one instance rather than the full collection that you were probably expecting.



If you need to work with the associated or embedded objects, then you might take the approach used by our example application. Lucene projection fetches the basic properties for all search results, including the entity object's primary key. When we later need to work with an entity object's associations, we use that primary key to retrieve only the necessary rows through a database call.

Faceted search

Lucene filters are a powerful tool for narrowing the scope of a query to some particular subset. However, filters work on predefined subsets. You must already know what it is that you are seeking.

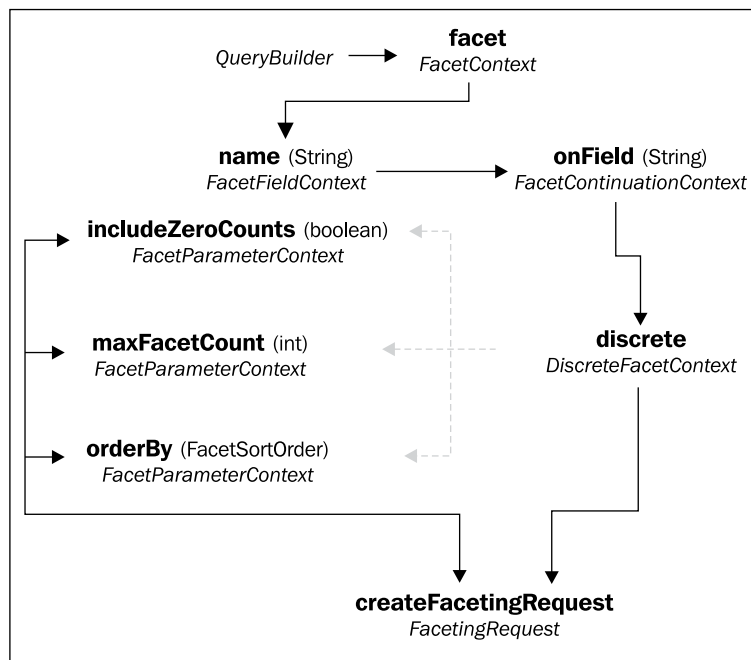
Sometimes you need to dynamically identify subsets. For example, let's give our `App` entity a `category` property representing its genre:

```
...
@Column
@Field
private String category;
...
```

When we perform a keyword search for apps, we might want to know which categories are represented in the results and how many results fall under each category. We might also want to know which price ranges were found. All of this information can help guide users in narrowing their queries more effectively.

Discrete facets

The process of dynamically identifying dimensions and then filtering by them is called **faceted search**. The Hibernate Search query DSL has a flow for this, starting with a `QueryBuilder` object's `facet` method:



Discrete faceting request flow (dotted gray arrows represent optional paths)

The `name` method takes some descriptive identifier for this facet (for example, `categoryFacet`), so that it can be referenced by queries later. The familiar `onField` clause declares the field by which to group results (for example, `category`).

The `discrete` clause indicates that we are grouping by single values, as opposed to ranges of values. We'll explore range facets in the next section.

The `createFacetingRequest` method completes this process and returns a `FacetingRequest` object. However, there are three optional methods that you can call first, in any combination:

- `includeZeroCounts`: It causes Hibernate Search to return all possible facets, even those which do not have any hits in the current search results. By default, facets with no hits are quietly ignored.
- `maxFacetCount`: It limits the number of facets to be returned.
- `orderBy`: It specifies the sort order of the facets found. The three options relevant to discrete facets are:
 - `COUNT_ASC`: Facets are sorted in an ascending order by the number of associated search results. The facets with the lowest number of hits are listed first.
 - `COUNT_DESC`: This is the exact opposite of `COUNT_ASC`. Facets are listed from the highest hit count to the lowest.
 - `FIELD_VALUE`: Facets are sorted in an alphabetical order by the value of the relevant field. For example, the "business" category would come before the "games" category.

This chapter's version of the VAPORware Marketplace now includes the following code for setting up a faceted search on the app category:

```
...
// Create a faceting request
FacetingRequest categoryFacetingRequest =
    queryBuilder
        .facet()
        .name("categoryFacet")
        .onField("category")
        .discrete()
        .orderBy(FacetSortOrder.FIELD_VALUE)
        .includeZeroCounts(false)
        .createFacetingRequest();

// Enable it for the FullTextQuery object
hibernateQuery.getFacetManager().enableFaceting(
    categoryFacetingRequest);
...
```

Now that the faceting request is enabled, we can run the search query and retrieve the facet information using the `categoryFacet` name that we just declared:

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

List<Facet> categoryFacets =
    hibernateQuery.getFacetManager().getFacets("categoryFacet");
...
```

The `Facet` class includes a `getValue` method, which returns the value of the field for a particular group. For example, if some of the matching apps are in the "business" category, then one of the facets will have the string "business" as its value. The `getCount` method reports how many search results are associated with that facet.

Using these two methods, our search servlet can iterate through all of the category facets, and build a collection to be used for display in the search results JSP:

```
...
Map<String, Integer> categories = new TreeMap<String, Integer>();
for(Facet categoryFacet : categoryFacets) {

    // Build a collection of categories, and the hit count for each
    categories.put(
        categoryFacet.getValue(), categoryFacet.getCount());

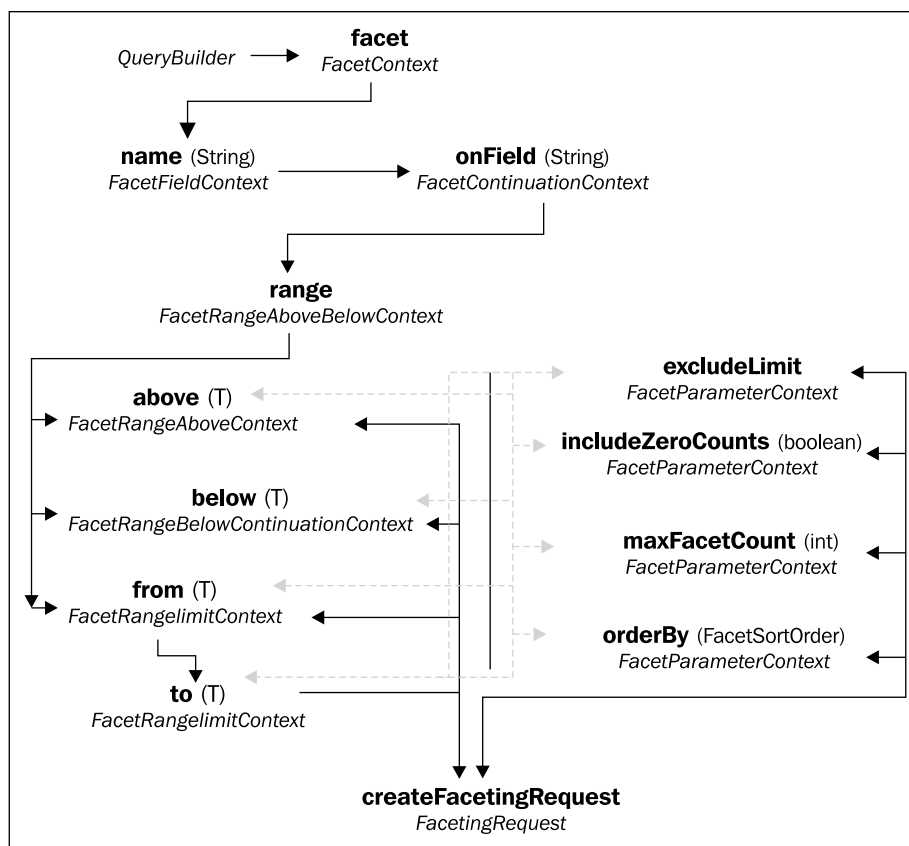
    // If this one is the *selected* category, then re-run the query
    // with this facet to narrow the results
    if(categoryFacet.getValue().equalsIgnoreCase(selectedCategory)) {
        hibernateQuery.getFacetManager()
            .getFacetGroup("categoryFacet").selectFacets(categoryFacet);
        apps = hibernateQuery.list();
    }
}
...
```

If the search servlet receives a request with a `selectedCategory` CGI parameter, then the user chooses to narrow results to a specific category. So if this string matches the value of a facet being iterated, then that facet is "selected" for the `FullTextQuery` object. The query can then be re-run, and it will then return only apps belonging to that category.

Range facets

Facets are not limited to single discrete values. A facet may also be created from a range of values. For example, we might want to group apps by a price range—search results priced below one dollar, between one and five dollars, or above five dollars.

The Hibernate Search DSL for range faceting takes the elements of the discrete faceting flow and combines them with elements from the range query that we saw in *Chapter 3, Performing Queries*:



Range faceting request flow (dotted gray arrows represent optional paths)

You can define a range as being above, below, or between two values (that is, `from - to`). These options may be used in combination to define as many range subsets as you wish.

As with regular range queries, the optional `excludeLimit` method exclude its boundary value from the range. In other words, `above(5)` means "greater than or equal to 5", whereas `above(5).excludeLimit()` means "greater than 5, *period*".

The optional `includeZeroCounts`, `maxFacetCount`, and `orderBy` methods operate in the same manner as with discrete faceting. However, range faceting offers an extra choice for sorting order. `FacetSortOrder.RANGE_DEFINITION_ORDER` causes facets to be returned in the order they were defined (note that the "r" is missing in "oder").

Along the discrete faceting request for `category`, the example code for this chapter also includes the following code snippet to enable range faceting for `price`:

```
...
FacetingRequest priceRangeFacetingRequest =
    queryBuilder
        .facet()
        .name("priceRangeFacet")
        .onField("price")
        .range()
        .below(1f).excludeLimit()
        .from(1f).to(5f)
        .above(5f).excludeLimit()
        .createFacetingRequest();
hibernateQuery.getFacetManager().enableFaceting(
    priceRangeFacetingRequest);
...
```

If you take a look at the source code for `search.jsp`, it now includes both the category and price range facets found during each search. These two faceting types may be used in combination to narrow the search results, with the currently-selected facets highlighted in bold. When **all** is selected for either type, that particular facet is removed and the search results widen again.

Categories	Price Range
business (4)	<u>above \$5</u>
<u>games</u> (2)	below \$1
<u>internet</u> (2)	<u>\$1 - \$5</u>
<u>media</u> (2)	<u>all</u>
<u>travel</u> (1)	
<u>all</u> (1)	

Query-time boosting

In *Chapter 3, Performing Queries*, we saw how to boost a field's relevance at index time, on either a fixed or a dynamic basis. It is also possible to dynamically change the weight at query time.

All query types in the Hibernate Search DSL include the `onField` and `andField` methods. For each query type, these two clauses also support a `boostedTo` method, taking a weight factor as a `float` parameter. Whatever the index-time weight of that field might be, adding a `boostedTo` clause multiplies it by the indicated number:

```
...
luceneQuery = queryBuilder
    .phrase()
    .onField("name").boostedTo(2)
    .andField("description").boostedTo(2)
    .andField("supportedDevices.name")
    .andField("customerReviews.comments")
    .sentence(unquotedSearchString)
    .createQuery();
...
```

In this chapter's version of the VAPORware Marketplace application, query-time boosting is now added to the "exact phrase" use case. When users wraps their search string in double quotes to search by phrase rather than by keywords, we want to give the `App` entity's name and description field even more weight than normal. The highlighted changes double the index-time weight of those two fields, but only for the exact phrase query rather than for all query types.

Placing time limits on a query

The example application we have been working with has a limited set of test data, only a dozen apps, and a handful of devices. So, as long as your computer has a reasonable amount of processor and memory resources, the search queries should run almost instantaneously.

However, an application with real data might involve searching across millions of entities, and there may be a risk of your queries taking too long. As a matter of user experience if nothing else, you will probably want to limit the execution of your queries to some reasonable period of time.

Hibernate Search offers two approaches for time boxing a query. One is through the `FullTextQuery` object's `limitExecutionTime` method:

```
...
hibernateQuery.limitExecutionTimeTo(2, TimeUnit.SECONDS);
...
```

This method causes the query to gracefully halt after a specified period of time, and return all of the results that it had found up until that point. The first parameter is the number of time units, and the second parameter is the type of time unit (for example, microsecond, millisecond, second, and so on). The preceding code snippet will try to stop the query after two seconds of searching.



After this query runs, you can determine whether or not it was interrupted by calling the object's `hasPartialResults()` method. This Boolean method returns `true` if the query timed out before reaching its natural conclusion.

The second approach, using the `setTimeout()` method, is similar in concept and in the parameters taken:

```
...
hibernateQuery.setTimeout(2, TimeUnit.SECONDS);
...
```

However, this method is for situations where the search should fail completely upon timeout, rather than proceeding as if it didn't happen. The preceding query object will throw a `QueryTimeoutException` exception after running for two full seconds, and will not return any results that were found during that time.



Be aware that with both of these approaches, Hibernate Search does the best it can to respect the specified period of time. It may actually take a bit more time for the query to halt.

Also, these timeout settings only affect Lucene access. Once your query has finished searching Lucene and starts pulling actual entities from the database, timeout control is in the hands of Hibernate ORM rather than Hibernate Search.

Summary

In this chapter, we explored more advanced techniques for narrowing search results, improving the quality of match relevance, and increasing performance.

We can now use Lucene filters to hone in on a fixed subset of matches. We have also seen how to use faceted search to dynamically identify subsets within results. Through projection-based queries, we can reduce or even eliminate the need for actual database calls. We now know how to adjust the relevance of fields at query time rather than at index time only. Last but not least, we are now able to set time limits on our queries and gracefully handle situations where a search runs too long.

In the next chapter, we will turn toward administration and maintenance, learning how to configure Hibernate Search and Lucene for optimal performance.

6

System Configuration and Index Management

In this chapter, we will look at configuration options for Lucene indexes, and learn how to perform basic maintenance tasks. We will see how to toggle between automatic and manual updates to Lucene indexes. We will examine low-latency write operations, synchronous versus asynchronous updates, and other performance tuning alternatives.

We will cover how to defragment and clean up a Lucene index for better performance, and how to use Lucene without touching hard drive storage at all. Last but not least, we will get exposure to the highly powerful **Luke** utility for working with Lucene indexes outside of application code.

Automatic versus manual indexing

So far, we really haven't had to think much about the timing of when entities are indexed. After all, Hibernate Search is tightly integrated with Hibernate ORM. By default, the add-on updates Lucene whenever the core updates the database.

However, you have the option of decoupling these operations, and indexing manually if you like. Some common situations where you might consider a manual approach are as follows:

- If you can easily live with Lucene being out of sync for limited periods, you might want to defer indexing operations until off-peak hours, to reduce system load during times of peak usage.
- If you want to use conditional indexing, but are not comfortable with the experimental nature of `EntityIndexingInterceptor` (refer to *Chapter 4, Advanced Mapping*), you might use manual indexing as an alternative approach.

- If your database may be updated directly, by processes that do not go through Hibernate ORM, you must manually update your Lucene indexes regularly to keep them in sync with the database.

To disable automatic indexing, set the `hibernate.search.indexing_strategy` property to `manual` in `hibernate.cfg.xml` (or `persistence.xml` if using JPA) as follows:

```
...  
<property name="hibernate.search.indexing_strategy">manual</property>  
...
```

Individual updates

When automatic indexing is disabled, manual indexing operations are driven by methods on a `FullTextSession` object (either the traditional Hibernate or the JPA version).

Adds and updates

The most important of these methods is `index`, which works with both **add** and **update** operations on the database side. This method takes one parameter, an instance of any entity class that is configured for Hibernate Search indexing.

This chapter's version of the VAPORware Marketplace application uses manual indexing. `StartupDataLoader` calls `index` for each app, immediately after persisting it in the database:

```
...  
fullTextSession.save(theCloud);  
fullTextSession.index(theCloud);  
...
```

On the Lucene side, the `index` method works within the same transactional context as the `save` method on the database side. The indexing only occurs when the transaction commits. In the event of a rollback, the Lucene index is untouched.



Using `index` manually overrides any conditional indexing rules. In other words, the `index` method ignores any `EntityIndexingInterceptor` that is registered with that entity class.

This is not the case for mass updates (see the *Mass updates* section), but is something to bear in mind when considering a manual indexing of individual objects. The code that calls `index` would be responsible for checking any conditions first.

Deletes

The basic method for removing an entity from a Lucene index is `purge`. This method is somewhat different from `index`, in that you do not pass it an object instance to remove. Instead, you pass it the class reference for the entity, and the ID of a particular instance to remove (that is, corresponding to `@Id` or `@DocumentId`):

```
...
fullTextSession.purge(App.class, theCloud.getId());
fullTextSession.delete(theCloud);
...
```

Hibernate Search also offers `purgeAll`, a convenient method for removing all the instances of a particular entity type. This method also takes the entity class reference, although obviously there is no need to pass a specific ID:

```
...
fullTextSession.purgeAll(App.class);
...
```

As with `index`, both `purge` and `purgeAll` operate within a transaction. **Deletes** do not actually occur until the transaction commits. Nothing happens in the event of a rollback.

If you *really* want to write to a Lucene index before the transaction commits, then the zero-parameter `flushToIndexes` method allows you to do so. This might be useful if you are processing a large number of entities, and want to free up memory along the way (with the `clear` method) to avoid `OutOfMemoryException`:

```
...
fullTextSession.index(theCloud);
fullTextSession.flushToIndexes();
fullTextSession.clear();
...
```

Mass updates

Adding, updating, and deleting entities individually can be rather tedious, and potentially error-prone if you miss things. Another option is to use `MassIndexer`, which can be thought of as a compromise of sorts between automatic and manual indexing.

This utility class is still instantiated and used manually. However, when it is called, it automatically rebuilds the Lucene indexes for all mapped entity classes in one step. There's no need to distinguish between adds, updates, and deletes, because the operation wipes out the entire index and recreates it from scratch.

A `MassIndexer` is instantiated with a `FullTextSession` object's `createIndexer` method. Once you have an instance, there are two ways to kick off the mass indexing:

- The `start` method indexes asynchronously, meaning that indexing occurs in a background thread while the flow of code in the main thread continues.
- The `startAndWait` method runs the indexing in synchronous mode, meaning that execution of the main thread is blocked until the indexing completes.

When running in synchronous mode, you need to wrap the operation with a try-catch block in case the main thread is interrupted while waiting:

```
...
try {
    fullTextSession.createIndexer().startAndWait();
} catch (InterruptedException e) {
    logger.error("Interrupted while waiting on MassIndexer: "
        + e.getClass().getName() + ", " + e.getMessage());
}
...
```



If practical, it is best to use mass indexing when the application is offline and not responding to queries. Indexing will place the system under heavy load, and Lucene will obviously be in a very inconsistent state relative to the database.

Mass indexing also differs from individual updates in two respects:

- A `MassIndexer` operation is not transactional. There is no need to wrap the operation within a Hibernate transaction, and likewise you cannot rely on a rollback if something goes wrong.
- `MassIndexer` does respect conditional indexing (refer to *Chapter 4, Advanced Mapping*). If you have an `EntityIndexingInterceptor` registered for that entity class, it will be invoked to determine whether or not to actually index particular instances.



`MassIndexer` support for conditional indexing was added in the 4.2 generation of Hibernate Search. If you are working with an application that uses an older version, you will need to migrate to 4.2 or higher in order to use `EntityIndexingInterceptor` and `MassIndexer` together.

Defragmenting an index

Changes to a Lucene index slowly make it less efficient over time, in the same way that a hard drive can become fragmented. When new entities are indexed, they go into a file (called a **segment**) that is separate from the main index file. When an entity is deleted, it actually remains in the index file and is simply marked as inaccessible.

These techniques help Lucene to keep its indexes as accessible for queries as possible, but it leads to slower performance over time. Having to open multiple segment files is slow, and can run up against operating system limits on the number of open files. Keeping deleted entities in the index makes the files more bloated than they need to be.

The process of merging all of these segments, and really purging deleted entities, is called **optimization**. It is analogous to defragmenting a hard drive. Hibernate Search provides mechanisms for optimizing your indexes on either on a manual or automatic basis.

Manual optimization

The `SearchFactory` class offers two methods for optimizing Lucene indexes manually. You can call these methods within your application, upon whatever event you like. Alternatively, you might expose them, and trigger your optimizations from outside the application (for example, with a web service called by a nightly cron job).

You can obtain a `SearchFactory` reference through a `FullTextSession` object's `getSearchFactory` method. Once you have an instance, its `optimize` method will defragment all available Lucene indexes:

```
...
fullTextSession.getSearchFactory().optimize();
...
```

Alternatively, you can use an overloaded version of `optimize`, taking an entity class as a parameter. This method limits the optimization to only that entity's Lucene index, as follows:

```
...
fullTextSession.getSearchFactory().optimize(App.class);
...
```



Another option is use a `MassIndexer` to rebuild your Lucene indexes (refer to the *Mass updates* section). Rebuilding an index from scratch leaves it in an optimized state anyway, so further optimization would be redundant if you are already performing that kind of maintenance regularly.

A *very* manual approach is to use the Luke utility, outside your application code altogether. See the section on Luke at the very end of this chapter.

Automatic optimization

An easier, if less flexible approach, is to have Hibernate Search trigger optimization for you automatically. This can be done on a global or a per-index basis. The trigger event can be a threshold number of Lucene changes, or a threshold number of transactions.

The chapter6 version of the VAPORware Marketplace application now contains the following four lines in its `hibernate.cfg.xml` file:

```
<property name="hibernate.search.default.optimizer.operation_limit.  
max">  
    1000  
</property>  
<property name="hibernate.search.default.optimizer.transaction_limit.  
max">  
    1000  
</property>  
<property name="hibernate.search.App.optimizer.operation_limit.max">  
    100  
</property>  
<property name="hibernate.search.App.optimizer.transaction_limit.max">  
    100  
</property>
```

The top two lines, referencing `default` in the property name, establish global defaults for all Lucene indexes. The last two lines, referencing `App`, are override values specific to the `App` entity.



Most of the configuration properties in this chapter may be made index-specific, by replacing the `default` substring with the name of the relevant index.

Normally this is the class name of the entity (for example, `App`), but it could be a custom name if you set the `index` element in that entity's `@Indexed` annotation.

Whether you deal at the global or index-specific level, `operation_limit.max` refers to a threshold number of Lucene changes (that is, adds or deletes). `transaction_limit.max` refers to a threshold number of transactions.

Overall, this snippet configures the `App` index for optimization after 100 transactions or Lucene changes. All other indexes will be optimized after 1,000 transactions or changes.

Custom optimizer strategy

You might enjoy the best of both worlds by using the automatic approach with a custom optimizer strategy. This chapter's version of the VAPORware Marketplace application uses a custom strategy to only allow optimization during off-peak hours. This custom class extends the default optimizer strategy, but only allows the base class to proceed with optimization when the current time is between midnight and 6:00 a.m.:

```
public class NightlyOptimizerStrategy
    extends IncrementalOptimizerStrategy {

    @Override
    public void optimize(Workspace workspace) {
        Calendar calendar = Calendar.getInstance();
        int hourOfDay = calendar.get(Calendar.HOUR_OF_DAY);
        if (hourOfDay >= 0 && hourOfDay <= 6) {
            super.optimize(workspace);
        }
    }
}
```



The easiest approach is to extend `IncrementalOptimizerStrategy`, and override the `optimize` method with your intercepting logic. However, if your strategy is fundamentally different from the default, then you can start with your own base class. Just have it implement the `OptimizerStrategy` interface.

To declare your own custom strategy, at either the global or per-index level, add a `hibernate.search.X.optimizer.implementation` property to `hibernate.cfg.xml` (where *X* is either *default*, or the name of a particular entity index):

```
...
<property name="hibernate.search.default.optimizer.implementation">
  com.packtpub.hibernatesearch.util.NightlyOptimizerStrategy
</property>
...
```

Choosing an index manager

An **index manager** is a component responsible for how and when changes are applied to a Lucene index. It coordinates the optimization strategy, the directory provider, and worker back ends (seen later in this chapter), and various other low-level components.

Hibernate Search includes two index manager implementations out of the box. The default is *directory-based*, and is a very sensible choice in most situations.

The other built-in alternative is *near-real-time*. It is a subclass inheriting from the *directory-based* index manager, but is designed for low-latency index writes. Rather than performing adds or deletes on the disk right away, this implementation queues them in the memory so they may be written more efficiently in batches.



The *near-real-time* implementation offers greater performance than the *directory-based* default, but there are two trade-offs. First, the *near-real-time* implementation is not available when using Lucene in a clustered environment (refer to *Chapter 7, Advanced Performance Strategies*). Secondly, because Lucene operations are not written to disk right away, they may be permanently lost in the event of an application crash.

As with most of the configuration properties covered in this chapter, an index manager may be selected on a global default or on a per-index basis. The difference is including *default*, or an entity index name (for example, *App*) in the property:

```
...
<property name="hibernate.search.default.indexmanager">
  directory-based
</property>
<property name="hibernate.search.App.indexmanager">
  near-real-time
</property>
...
```

It is possible to write your own index manager implementation. To get a deeper sense of how index managers function, review the source code of the two implementations provided out of the box. The directory-based manager is implemented by `DirectoryBasedIndexManager`, and the near-real-time manager by `NRTIndexManager`.



An easy approach to writing a custom implementation is to subclass one of the two built-in options, and override methods only as needed. If you want to create a custom index manager completely from scratch, then it would need to implement the `org.hibernate.search.indexes.spi.IndexManager` interface.

Applying a custom index manager, at the global or the per-index level, works the same as the built-in options. Just set the appropriate property to your implementation's fully qualified class name (for example, `com.packtpub.hibernatesearch.util.MyIndexManager`) rather than the directory-based or near-real-time strings.

Configuring workers

One of the component types that index managers coordinate are **workers**, which are responsible for the actual updates made to a Lucene index.

If you are using Lucene and Hibernate Search in a clustered environment, many of the configuration options are set at the worker level. We will explore those more fully in *Chapter 7, Advanced Performance Strategies*. However, three key configuration options are available in any environment.

Execution mode

By default, workers perform Lucene updates **synchronously**. That is, once an update begins, execution of the main thread is blocked until that update completes.

Workers may instead be configured to update **asynchronously**, a "fire and forget" mode that spawns a separate thread to perform the work. The advantages are that the main thread will be more responsive, and the workload handled more efficiently. The downside is that the database and the index may be out of sync for very brief periods.


Execution mode is declared in `hibernate.cfg.xml` (or `persistence.xml` for JPA). A global default may be established with the `default` substring, and per-entity configurations may be set with the entity index name (for example, `App`):

```
...
<property name="hibernate.search.default.worker.execution">
    sync
</property>
<property name="hibernate.search.App.worker.execution">
    async
</property>
...
```

Thread pool

By default workers perform updates in only one thread, either the main thread in the synchronous mode, or a single spawned thread in the asynchronous mode. However, you have the option of creating a larger pool of threads to handle the work. The pool may apply at the global default level, or be specific to a particular index:

```
...
<property name="hibernate.search.default.worker.thread_pool.size">
    2
</property>
<property name="hibernate.search.App.worker.thread_pool.size">
    5
</property>
...
```

 Because of the way that Lucene indexes are locked during update operations, using a lot of threads in parallel often does not provide the performance boost that you might expect. However, it is worth experimenting when tuning and load-testing an application.

Buffer queue

Pending work gets backed up in a queue, waiting for a thread to free up and deal with it. By default, the size of this buffer is infinite, at least in theory. In reality, it is bound by the amount of system memory available, and an `OutOfMemoryException` may be thrown if the buffer grows too large.

Therefore, it is a good idea to place some limit, globally or on a per-index basis, for the size to which these buffer can grow.

```
...
<property name="hibernate.search.default.worker.buffer_queue.max">
    50
</property>
<property name="hibernate.search.App.worker.buffer_queue.max">
    250
</property>
...
```

When a buffer reaches the maximum allowable size for its index, additional operations will be performed by the thread which creates them. This blocks execution and slows down performance, but ensures that the application will not run out of memory. Experiment to find a balanced threshold for an application.

Selecting and configuring a directory provider

Both of the built-in index managers use a subclass `DirectoryBasedIndexManager`. As the name implies, both of them make use of Lucene's abstract class `Directory`, to manage the form in which indexes are stored.

In the *Chapter 7*, we will look at some special directory implementations geared for clustered environments. However, in single-server environments the two built-in choices are filesystem storage, and storage in memory.

Filesystem-based

By default, Lucene indexes are stored on the filesystem, in the current working directory of the Java application. No configuration is necessary for this arrangement, but it has been explicitly set in all versions of the VAPORware Marketplace application so far with this property in `hibernate.cfg.xml` (or `persistence.xml`):

```
...
<property name="hibernate.search.default.directory_provider">
    filesystem
</property>
...
```

As with the other configuration properties that we've seen in this chapter, you could replace `default` with a particular index name (for example, `App`).

When using filesystem-based indexes, you probably want to use a known fixed location rather than the current working directory. You can specify either a relative or absolute path with the `indexBase` property. In all of the VAPORware Marketplace versions that we've seen so far, the Lucene indexes have been stored under each Maven project's `target` directory, so that Maven removes them up before each fresh build:

```
...
<property name="hibernate.search.default.indexBase">
    target/lucenceIndex
</property>
...
```

Locking strategy

All Lucene directory implementations lock their indexes when writing to them, to prevent corruption from multiple processes or threads writing to them simultaneously. There are four locking strategies available, and you can specify one by setting the `hibernate.search.default.locking_strategy` property to one of these strings:

- `native`: This is the default strategy for filesystem-based directories, when no locking strategy property is specified. It relies on file locking at the native operating system level, so that if your application crashes the index locks will still be released. However, the downside is that this strategy should not be used when your indexes are stored remotely on a network shared drive.
- `simple`: This strategy relies on the JVM to handle file locking. It is safer to use when your Lucene index is on a remote shared drive, but locks will not be cleanly released if the application crashes or has to be killed.
- `single`: This strategy does not create a lock file on the filesystem, but rather uses a Java object in memory (similar to a `synchronized` block in multithreaded Java code). For a single-JVM application, this works well no matter where the index files are, and there is no issue with locks being released after a crash. However, this strategy is only viable if you are sure that no other process outside the JVM might write to your index files.
- `none`: It does not use locking at all. This is *not* a recommended option.



To remove locks that were not cleanly released, use the Luke utility explored in the *Using the Luke utility* section of this chapter.

RAM-based

For testing and demo purposes, our VAPORware Marketplace application has used an in-memory H2 database throughout this book. It is recreated every time the application starts, and is destroyed when the application stops, with nothing being persisted to permanent storage along the way.

Lucene indexes are able to work in the exact same manner. In this chapter's version of the example application, the `hibernate.cfg.xml` file has been modified to store its index in RAM rather than on the filesystem:

```
...
<property name="hibernate.search.default.directory_provider">
    ram
</property>
...
```




The RAM-based directory provider initializes its Lucene indexes when the `Hibernate SessionFactory` (or `JPA EntityManagerFactory`) is created. Be aware that when you close this factory, it destroys all your indexes!

This shouldn't be a problem when using a modern dependency-injection framework, because the framework will keep your factory in memory and available when needed. Even in our vanilla example application, we have stored a singleton `SessionFactory` in the `StartupDataLoader` class for this reason.

An in-memory index would seem to offer greater performance, and it may be worth experimenting with in your application tuning. However, it is not generally recommended to use the RAM-based directory provider in production settings.

First and foremost, it is easy to run out of memory and crash the application with a large data set. Also, your application has to rebuild its indexes from scratch upon each and every restart. Clustering is not an option, because only the JVM which created the in-memory index has access to that memory. Last but not least, the filesystem-based directory provider already makes intelligent use of caching, and its performance is surprisingly comparable to the RAM-based provider.

All that being said, the RAM-based provider is a common approach for testing applications. Unit tests are likely to involve fairly small sets of data, so running out of memory is not a concern. Also, having the indexes completely and cleanly destroyed in between each unit test might be more of a feature than a drawback.

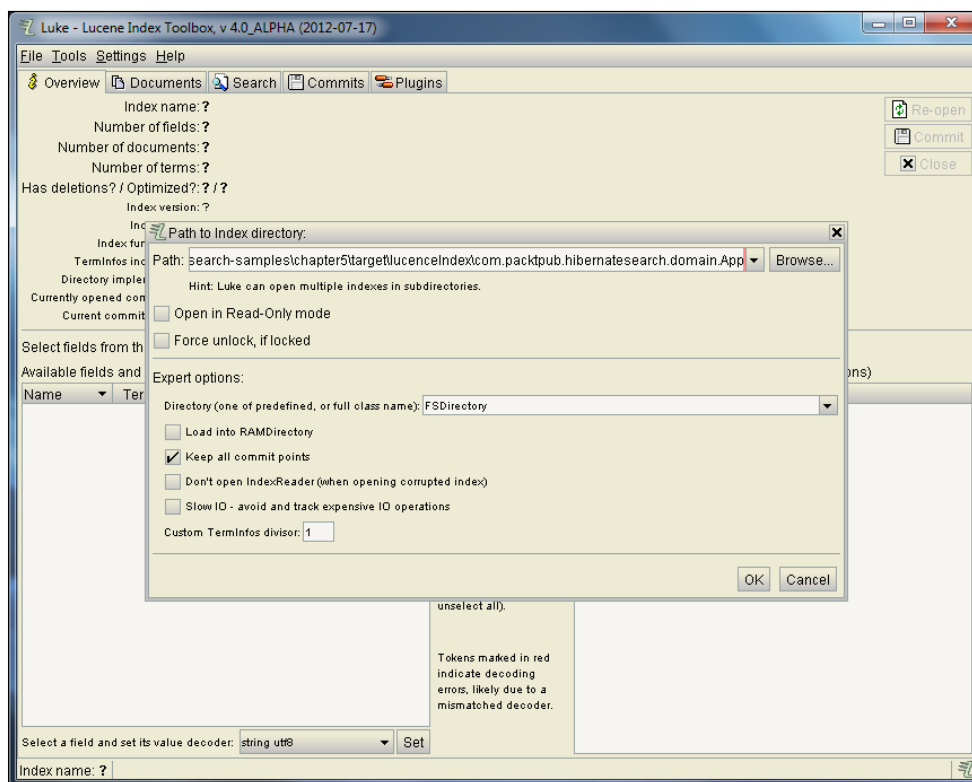
 The RAM-based directory provider defaults to the single locking strategy, and it really makes no sense to change this.

Using the Luke utility

Hibernate ORM gives your application code pretty much everything it needs to interact with the database. However, you probably still use some sort of SQL client to manually work with your database outside the context of your application code.

Likewise, it can be useful to explore a Lucene index manually without having to write code for the task. Luke (<http://code.google.com/p/luke>) is a very useful utility that fills this role for Lucene. You can use Luke to browse your indexes, test queries, and perform helpful tasks such as removing index locks that did not cleanly release.

The Luke download is a monolithic executable JAR file. Double-clicking the JAR, or otherwise executing it from a console prompt, brings up a graphical interface and a prompt for your index location, as shown in the following screenshot:



The previous screenshot shows Luke at startup. Unfortunately, Luke can only access filesystem-based indexes, not the RAM-based index used in this chapter. So in these examples, Luke points to the `chapter5` code file directory's Maven project workspace. The `App` entity index is located under `target/luceneIndex/com.packtpub.hibernatesearch.domain.App`.

Notice the **Force unlock, if locked** checkbox near the top of the open-index dialog box. If you have an index for which a file lock did not cleanly release (refer to the *Locking strategy* section), then you can fix the problem by checking this box and opening the index.

Once you have opened a Lucene index, Luke displays an assortment of information about the number of indexed documents (that is, entities), the current state of optimization (that is, fragmentation), and other details, as shown in the following screenshot:

Index name: `E:\eclipse\workspace\hibernate-search-samples\chapter5\target\luceneIndex\com.packtpub.hibernatesearch.domain.App`

Number of fields: **14**
 Number of documents: **11**
 Number of terms: **624**
 Has deletions? / Optimized?: **No / Yes**
 Index version: 13af0ceb63f
 Index format: Lucene 4.0
 Index functionality: flexible, codec-specific
 TermInfos index divisor: 1
 Directory implementation: org.apache.lucene.store.MMapDirectory
 Currently opened commit point: segments_3 (generation=3, segs=1)
 Current commit user data: --

Select fields from the list below, and press button to view top terms in these fields. No selection means all fields.

Available fields and term counts per field:

Name	Term count	%	Decoder
_hibernate_c	1	0.16 %	string utf8
category	5	0.8 %	string utf8
customerRev	115	18.43 %	string utf8
customerRev	12	1.92 %	string utf8
description	364	58.33 %	string utf8
id	11	1.76 %	string utf8
image	11	1.76 %	string utf8
name	26	4.17 %	string utf8
price	48	7.69 %	string utf8
releaseDate	1	0.16 %	string utf8
sorting_name	11	1.76 %	string utf8
supportedDe	5	0.8 %	string utf8
supportedDe	4	0.64 %	string utf8
supportedDe	10	1.6 %	string utf8

Number of top terms: 50

Hint: use Shift-Click to select ranges, or Ctrl-Click to select multiple fields (or unselect all).

Tokens marked in red indicate decoding errors, likely due to a mismatched decoder.

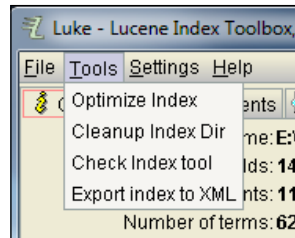
Top ranking terms. (Right-click for more options)

Rank	Freq	Field	Text
1	11	_hibernate_c	com.packtpub.hibernatesearch
2	11	customerRev	h
3	11	customerRev	d@
4	11	customerRev	
5	11	description	AP
6	11	description	app
7	11	releaseDate	20121114
8	11	customerRev	p
9	11	customerRev	l
10	11	customerRev	x
11	11	customerRev	t
12	10	description	AR
13	9	supportedDe	orange
14	9	description	your
15	8	supportedDe	song
16	8	supportedDe	xphone

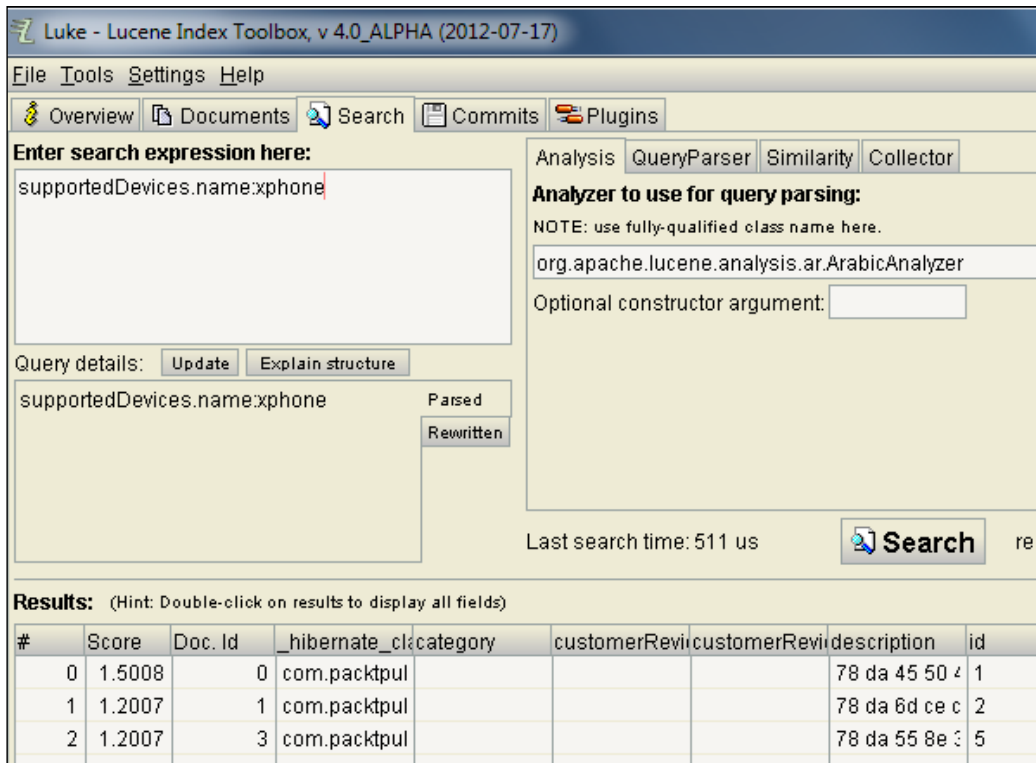
Select a field and set its value decoder: string utf8 Set

Index name: `E:\eclipse\workspace\hibernatesearch.domain.App`

From the **Tools** menu at the top of the utility, you have options for performing basic maintenance tasks such as checking the index for corruption, or manual optimization (that is, defragmenting). These operations are best performed during off-peak hours, or during a full outage window.



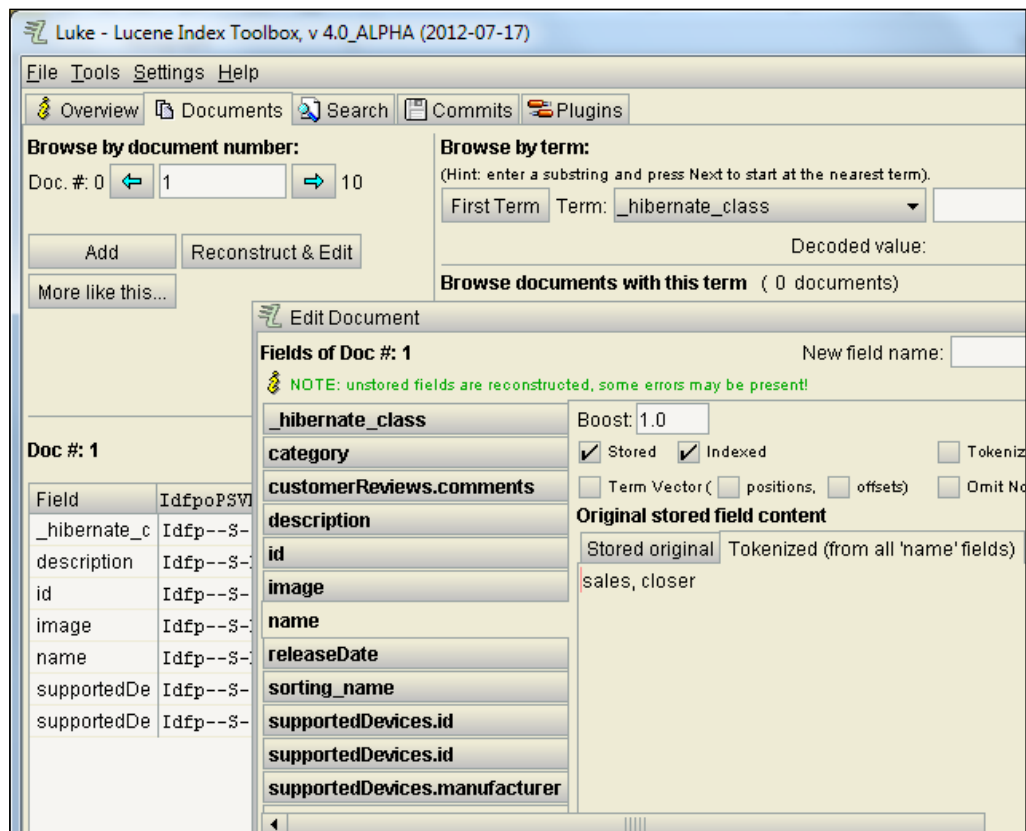
The **Documents** tab allows you to browse through entities one by one, which may have some limited use. Much more interesting is the **Search** tab, which allows you to explore your index using free-form Lucene queries, as shown in the following screenshot:



The full-blown Lucene API is beyond the scope of this book, but here are some basics to get you started:

- Search expressions are in the form of a field name and a desired value, separated by a colon. For example, to search for apps in the business category, use the search expression `category:business`.
- Associated items may be specified with the entity field name, followed by a period, followed by the field name within the associated item. In the above screenshot, we are searching for all apps supported on the xPhone device, by using the search expression `supportedDevices.name:xphone`.
- Remember that the default analyzer converts terms into lower case during the indexing process. So if you wanted to search on xPhone for example, be sure to type it as `xphone`.

If you double-click on one of the search results found, then Luke flips over to the **Documents** tab with the relevant document loaded. Click on the **Reconstruct & Edit** button to examine that entity's fields, as shown in the following screenshot:



Browsing this data will give you a feel for how the analyzer parses your entities. Words will be filtered out, and text will be tokenized unless you configured the `@Field` annotation to the contrary (as we did with `sorting_name`). If a Hibernate Search query doesn't return the results that you expect, browsing field data in Luke can help you spot the issue.

Summary

In this chapter we saw how to update Lucene indexes manually, one entity object at a time or in bulk, as an alternative to letting Hibernate Search manage updates automatically. We learned about the fragmentation that accumulates from Lucene update operations, and how to approach optimization on a manual or automatic basis.

We explored various performance tuning options for Lucene, from low-latency writes to multi-threaded asynchronous updates. We now know how to configure Hibernate Search for creating Lucene indexes on either the filesystem or RAM, and why you might choose one over the other. Finally, we worked with the Luke utility to inspect and perform maintenance tasks on a Lucene index without having to go through an application's Hibernate Search code.

In the next chapter, we will look at some advanced strategies for improving the performance of your applications. This will include recapping the performance tips covered so far, before diving into server clusters and Lucene index sharding.

7

Advanced Performance Strategies

In this chapter, we will look at some advanced strategies for improving the performance and scalability of production applications, through code as well as server architecture. We will explore options for running applications in multi-node server clusters, to spread out and handle user requests in a distributed fashion. We will also learn how to use sharding to help make our Lucene indexes faster and more manageable.

General tips

Before diving into some advanced strategies for improving performance and scalability, let's briefly recap some of the general performance tips already spread across the book:

- When mapping your entity classes for Hibernate Search, use the optional elements of the `@Field` annotation to strip the unnecessary bloat from your Lucene indexes (see *Chapter 2, Mapping Entity Classes*):
 - If you are definitely not using index-time boosting (see *Chapter 4, Advanced Mapping*), then there is no reason to store the information needed to make this possible. Set the `norms` element to `Norms.NO`.
 - By default, the information needed for a projection-based query is not stored unless you set the `store` element to `Store.YES` or `Store.COMPRESS` (see *Chapter 5, Advanced Querying*). If you had projection-based queries that are no longer being used, then remove this element as part of the cleanup.

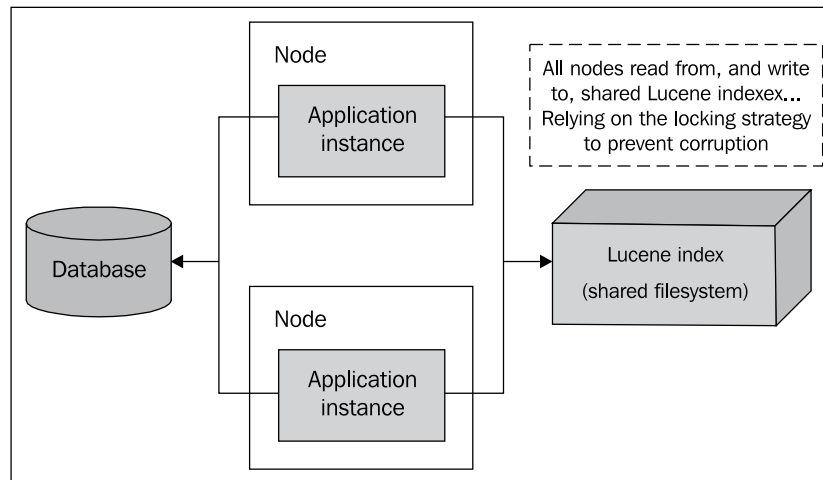
- Use conditional indexing (see *Chapter 4, Advanced Mapping*) and partial indexing (*Chapter 2, Mapping Entity Classes*) to reduce the size of Lucene indexes.
- Rely on filters to narrow your results at the Lucene level, rather than using a WHERE clause at the database query level (see *Chapter 5, Advanced Querying*).
- Experiment with projection-based queries wherever possible (see *Chapter 5, Advanced Querying*), to reduce or eliminate the need for database calls. Be aware that with advanced database caching, the benefits might not always justify the added complexity.
- Test various index manager options (see *Chapter 6, System Configuration and Index Management*), such as trying the near-real-time index manager or the async worker execution mode.

Running applications in a cluster

Making modern Java applications scale in a production environment usually involves running them in a cluster of server instances. Hibernate Search is perfectly at home in a clustered environment, and offers multiple approaches for configuring a solution.

Simple clusters

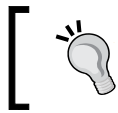
The most straightforward approach requires very little Hibernate Search configuration. Just set up a file server for hosting your Lucene indexes and make it available to every server instance in your cluster (for example, NFS, Samba, and so on):



A simple cluster with multiple server nodes using a common Lucene index on a shared drive

Each application instance in the cluster uses the default index manager, and the usual `filesystem` directory provider (see *Chapter 6, System Configuration and Index Management*).

In this arrangement, all of the server nodes are true peers. They each read from the same Lucene index, and no matter which node performs an update, that node is responsible for the write. To prevent corruption, Hibernate Search depends on simultaneous writes being blocked, by the locking strategy (that is, either "simple" or "native", see *Chapter 6, System Configuration and Index Management*).



Recall that the "near-real-time" index manager is explicitly incompatible with a clustered environment.

The advantage of this approach is two-fold. First and foremost is simplicity. The only steps involved are setting up a filesystem share, and pointing each application instance's directory provider to the same location. Secondly, this approach ensures that Lucene updates are instantly visible to all the nodes in the cluster.

However, a serious downside is that this approach can only scale so far. Very small clusters may work fine, but larger numbers of nodes trying to simultaneously access the same shared files will eventually lead to lock contention.

Also, the file server on which the Lucene indexes are hosted is a single point of failure. If the file share goes down, then your search functionality breaks catastrophically and instantly across the entire cluster.

Master-slave clusters

When your scalability needs outgrow the limitations of a simple cluster, Hibernate Search offers more advanced models to consider. The common element among them is the idea of a master node being responsible for all Lucene write operations.

Clusters may also include any number of slave nodes. Slave nodes may still initiate Lucene updates, and the application code can't really tell the difference. However, under the covers, slave nodes delegate that work to be actually performed by the master node.

Directory providers

In a master-slave cluster, there is still an "overall master" Lucene index, which logically stands apart from all of the nodes. This may be filesystem-based, just as it is with a simple cluster. However, it may instead be based on JBoss Infinispan (<http://www.jboss.org/infinispan>), an open source in-memory NoSQL datastore sponsored by the same company that principally sponsors Hibernate development:

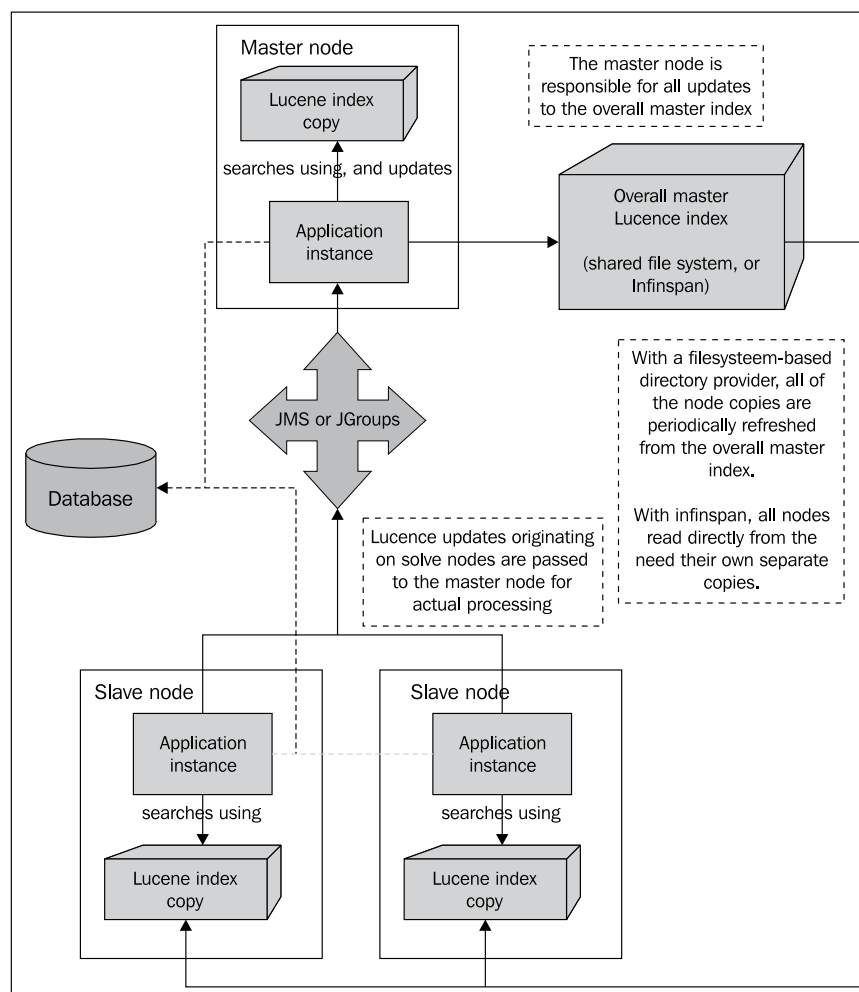
- In a **filesystem-based** approach, all nodes keep their own local copies of the Lucene indexes. The master node actually performs updates on the overall master indexes, and all of the nodes periodically read from that overall master to refresh their local copies.
- In an **Infinispan-based** approach, the nodes all read from the Infinispan index (although it is still recommended to delegate writes to a master node). Therefore, the nodes do not need to maintain their own local index copies. In reality, because Infinispan is a distributed datastore, portions of the index will reside on each node anyway. However, it is still best to visualize the overall index as a separate entity.

Worker backends

There are two available mechanisms by which slave nodes delegate write operations to the master node:

- A **JMS** message queue provider creates a queue, and slave nodes send messages to this queue with details about Lucene update requests. The master node monitors this queue, retrieves the messages, and actually performs the update operations.
- You may instead replace JMS with **JGroups** (<http://www.jgroups.org>), an open source multicast communication system for Java applications. This has the advantage of being faster and more immediate. Messages are received in real-time, synchronously rather than asynchronously.

However, JMS messages are generally persisted to a disk while awaiting retrieval, and therefore can be recovered and processed later, in the event of an application crash. If you are using JGroups and the master node goes offline, then all the update requests sent by slave nodes during that outage period will be lost. To fully recover, you would likely need to reindex your Lucene indexes manually.



A master-slave cluster using a directory provider based on filesystem or Infinispan, and worker based on JMS or JGroups. Note that when using Infinispan, nodes do not need their own separate index copies.

A working example

Experimenting with all of the possible clustering strategies requires consulting the Hibernate Search Reference Guide, as well as the documentation for Infinispan and JGroups. However, we will get started by implementing a cluster with the filesystem and JMS approach, since everything else is just a variation on this standard theme.

This chapter's version of the VAPORware Marketplace application discards the Maven Jetty plugin that we've been using all along. This plugin is great for testing and demo purposes, but it is meant for running a single server instance, and we now need to run at least two Jetty instances simultaneously.

To accomplish this, we will configure and launch Jetty instances programmatically. If you look under `src/test/java/` in the `chapter7` project, there is now a `ClusterTest` class. It is structured for JUnit 4, so that Maven can automatically invoke its `testCluster()` method after a build. Let's take a look at the relevant portions of that test case method:

```
...
String projectBaseDirectory = System.getProperty("user.dir");
...
Server masterServer = new Server(8080);
WebAppContextmasterContext = new WebAppContext();
masterContext.setDescriptor(projectBaseDirectory +
    "/target/vaporware/WEB-INF/web.xml");
...
masterServer.setHandler(masterContext);
masterServer.start();
...
Server slaveServer = new Server(8181);
WebAppContextslaveContext = new WebAppContext();
slaveContext.setDescriptor(projectBaseDirectory +
    "/target/vaporware/WEB-INF/web-slave.xml");
...
slaveServer.setHandler(slaveContext);
slaveServer.start();
...
```

Although this is all running on one physical machine, we are simulating a cluster for test and demo purposes. One Jetty server instance launches on port 8080 as the master node, and another Jetty server launches on port 8181 as a slave node. The difference between the two nodes is that they use separate `web.xml` files, which in turn load different listeners upon startup.

In the previous versions of this application, a `StartupDataLoader` class handled all of the database and Lucene initialization. Now, the two nodes use `MasterNodeInitializer` and `SlaveNodeInitializer`, respectively. These in turn load Hibernate ORM and Hibernate Search settings from separate files, named `hibernate.cfg.xml` and `hibernate-slave.cfg.xml`.



There are many ways in which you might configure an application for running as the master node or as a slave node instance. Rather than building separate WARs, with separate versions of `web.xml` or `hibernate.cfg.xml`, you might use a dependency injection framework to load the correct settings based on something in the environment.

Both versions of the Hibernate the config file set the following Hibernate Search properties:

- `hibernate.search.default.directory_provider`: In previous chapters we have seen this populated with either `filesystem` or `ram`. The other option discussed earlier is `infinispan`.

Here, we use `filesystem-master` and `filesystem-slave` on the master and slave node, respectively. Both of these directory providers are similar to regular `filesystem`, and work with all of the related properties that we've seen so far (e.g. location, locking strategy, etc).

However, the "master" variant includes functionality for periodically refreshing the overall master Lucene indexes. The "slave" variant does the reverse, periodically refreshing its local copy with the overall master contents.

- `hibernate.search.default.indexBase`: Just as we've seen with single-node versions in the earlier chapters, this property contains the base directory for the *local* Lucene indexes. Since our example cluster here is running on the same physical machine, the master and slave nodes use different values for this property.
- `hibernate.search.default.sourceBase`: This property contains the base directory for the *overall master* Lucene indexes. In a production setting, this would be on some sort of shared filesystem, mounted and accessible to all nodes. Here, the nodes are running on the same physical machine, so the master and slave nodes use the same value for this property.
- `hibernate.search.default.refresh`: This is the interval (in seconds) between index refreshes. The master node will refresh the overall master indexes after each interval, and slave nodes will use the overall master to refresh their own local copies. This chapter's version of the VAPORware Marketplace application uses a 10-second setting for demo purposes, but that would be far too short for production. The default setting is 3600 seconds (one hour).

To establish a JMS worker backend, there are three additional settings required for the slave node *only*:

- `hibernate.search.default.worker.backend`: Set this value to `jms`. The default value, `lucene`, has been applied in earlier chapters because no setting was specified. If you use JGroups, then it would be set to `jgroupsMaster` or `jgroupsSlave` depending upon the node type.

- `hibernate.search.default.worker.jms.connection_factory`: This is the name by which Hibernate Search looks up your JMS connection factory in JNDI. This is similar to how Hibernate ORM uses the `connection.datasource` property to retrieve a JDBC connection from the database.

In both the cases, the JNDI configuration is specific to the app server in which your application runs. To see how the JMS connection factory is set up, see the `src/main/webapp/WEB-INF/jetty-env.xml` Jetty configuration file. We are using Apache ActiveMQ in this demo, but any JMS-compatible provider would work just as well.

- `hibernate.search.default.worker.jms.queue`: The JNDI name of the JMS queue to which slave nodes send write requests to Lucene. This too is configured at the app server level, right alongside the connection factory.

With these worker backend settings, a slave node will automatically send a message to the JMS queue that a Lucene update is needed. To see that this is happening, the new `MasterNodeInitializer` and `SlaveNodeInitializer` classes each load half of the usual test data set. We will know that our cluster works if all of the test entities are eventually indexed together, and are being returned by search queries that are run from either nodes.

Although Hibernate Search sends messages from the slave nodes to the JMS queue automatically, it is your responsibility to have the master node retrieve those messages and process them.

In a JEE environment, you might use a message-driven bean, as is suggested by the Hibernate Search documentation. Spring also has a task execution framework that can be leveraged. However, in any framework, the basic idea is that the master node should spawn a background thread to monitor the JMS queue and process its messages.

This chapter's version of the VAPORware Marketplace application contains a `QueueMonitor` class for this purpose, which is wrapped into a `Thread` object and spawned by the `MasterNodeInitializer` class.

To perform the actual Lucene updates, the easiest approach is to create your own custom subclass of `AbstractJMSHibernateSearchController`. Our implementation is called `QueueController`, and does little more than wrapping this abstract base class.

When the queue monitor receives a `javax.jms.Message` object from the JMS queue, it is simply passed as-is to the controller's base class method `onMessage`. That built-in method handles the Lucene update for us.



As you can see, there is a lot more involved to a master-slave clustering approach than there is to a simple cluster. However, the master-slave approach offers a dramatically greater upside in scalability.

It also reduces the single-point-of-failure risk. It is true that this architecture involves a single "master" node, through which all Lucene write operations must flow. However, if the master node goes down, the slave nodes continue to function, because their search queries run against their own local index copies. Also, update requests should be persisted by the JMS provider, so that those updates can still be performed once the master node is brought back online.

Because we are spinning up Jetty instances programmatically, rather than through the Maven plugin, we pass a different set of goals to each Maven build. For the `chapter7` project, you should run Maven as follows:

```
mvn clean compile war:exploded test
```

You will be able to access the "master" node at `http://localhost:8080`, and the "slave" node at `http://localhost:8181`. If you are very quick about firing off a search query on the master node the moment it starts, then you will see it returning only half of the expected results! However, within a few seconds, the slave node updates arrive through JMS. Both the halves of the data set will merge and be available across the cluster.

Sharding Lucene indexes

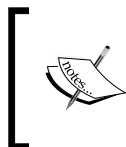
Just as you can balance your application load across multiple nodes in a cluster, you may also split up your Lucene indexes through a process called **sharding**. You might consider sharding for performance reasons if your indexes grow to a very large size, as larger index files take longer to index and optimize than smaller shards.

Sharding may offer additional benefits if your entities lend themselves to partitioning (for example, by language, geography, and so on). Performance may be improved if you can predictably steer queries toward the specific appropriate shard. Also, it sometimes makes lawyers happy when you can store "sensitive" data at a physically different location.

Even though its dataset is very small, this chapter's version of the VAPORware Marketplace application now splits its App index into two shards. The relevant line in `hibernate.cfg.xml` looks similar to the following:

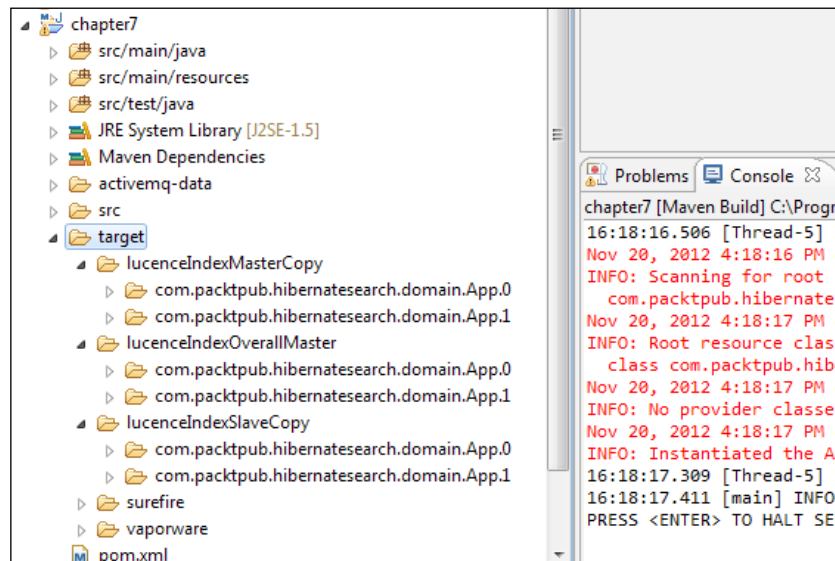
```
...
<property
  name="hibernate.search.default.sharding_strategy.nbr_of_shards">
    2
  </property>
...
```

As with all of the other Hibernate Search properties that include the substring `default`, this is a global setting. It can be made index-specific by replacing `default` with an index name (for example, `App`).



This exact line appears in both `hibernate.cfg.xml` (used by our "master" node), and `hibernate-slave.cfg.xml` (used by our "slave" node). When running in a clustered environment, your sharding configuration should match all the nodes.

When an index is split into multiple shards, each shard includes the normal index name followed by a number (starting with zero). For example, `com.packtpub.hibernatesearch.domain.App.0` instead of just `com.packtpub.hibernatesearch.domain.App`. This screenshot shows the Lucene directory structure of our two-node cluster, while it is up and running with both nodes configured for two shards:



An example of sharded Lucene indexes running in a cluster (note the numbering of each App entity directory)

Just as the shards are numbered on the filesystem, they can be separately configured by number in `hibernate.cfg.xml`. For example, if you want to store the shards at different locations, you might set properties as follows:

```
...
<property name="hibernate.search.App.0.indexBase">
    target/lucenceIndexMasterCopy/EnglishApps
</property>
<property name="hibernate.search.App.1.indexBase">
    target/lucenceIndexMasterCopy/FrenchApps
</property>
...
```

When a Lucene write operation is performed for an entity, or when a search query needs to read from an entity's index, a **sharding strategy** determines which shard to use.

If you are sharding simply to reduce the file size, then the default strategy (implemented by `org.hibernate.search.store.impl.IdHashShardingStrategy`) is perfectly fine. It uses each entity's ID to calculate a unique hash code, and distributes the entities among the shards in a roughly even manner. Because the hashing calculation is reproducible, the strategy is able to direct future updates for an entity towards the appropriate shard.

To create your own custom sharding strategy with more exotic logic, you can create a new subclass inheriting from `IdHashShardingStrategy`, and tweak it as needed. Alternatively, you can completely start from scratch with a new class implementing the `org.hibernate.search.store.IndexShardingStrategy` interface, perhaps referring to the source code of `IdHashShardingStrategy` for guidance.

Summary

In this chapter, we learned how to work with applications in a modern distributed server architecture, to allow for scalability and better performance. We have seen a working cluster implemented with a filesystem-based directory provider and JMS-based backend, and now have enough knowledge to explore other approaches involving Infinispan and JGroups. We used sharding to split a Lucene index into smaller chunks, and know how to go about implementing our own custom sharding strategy, if necessary.

This brings us to the end of our little adventure with Hibernate Search! We have covered a lot of critical concepts about Hibernate, Lucene and Solr, and searches in general. We have learned how to map our data to search indexes, query and update those indexes at runtime, and arrange it all in the best architecture for a given project. We did all of this through an example application, that grew with our knowledge from simple to advanced along the way.

There's always more to learn. Hibernate Search can work with dozens of Solr components for more advanced functionality, and integrating with a new generation of "NoSQL" data stores is possible as well. However, you are now equipped with enough core knowledge to explore these horizons independently, if you wish. Until next time, thank you for reading! You can find me online at steveperkins.net, and I would love to hear from you.

Index

Symbols

- @Analyzer** annotation 72
- @AnalyzerDef** annotation 71, 72
- @AnalyzerDiscriminator** annotation 74
- @ClassBridge** annotation 67
- @ContainedIn** annotation 34
- @DynamicBoost** annotation 75, 76
- @ElementCollection** annotation 37, 38
- @Factory** annotation 82
- @Field** annotation 30, 35, 37, 60
- @FieldBridge** annotation 62
- @FullTextFilterDefs** annotation 85
- @GeneratedValue** field 10
- @Indexed** annotation 10, 32, 76
- @IndexedEmbedded** annotation 34
- @ManyToMany** annotation 34, 37
- @NumericField** annotation 31
- @WebListener** annotation 11
- @WebServlet** annotation maps 15

A

- active variable** 77
- add and update operation, individual updates** 100
- AliasToBeanResultTransformer** subclass 88
- All Devices** option 86
- analysis** 68
- analyze** 30
- analyzer definition**
 - about 70
 - dynamic analyzer, selecting 72-74
 - static analyzer, selecting 71, 72
- andField** method 48, 51

- Apache Commons Database Connection Pools**

- URL 20

API

- selecting, for Hibernate ORM 27, 28

- App** class 8, 32

application

- Hibernate Search, incorporating 7
- running 21-26

- asterisk (*)** 50

automatic indexing

- disabling 100
- versus manual indexing 99, 100

- automatic optimization** 104

B

- bidirectional bridge** 63

- Boolean (combination) queries** 53

- bool** method 53

- boostedTo** method 95

boosting

- dynamic boosting, at index time 75, 76
- static boosting, at index time 74, 75

- buffer queue, workers** 108

build system

- selecting 17, 18

C

- cascade element** 34

character filtering

- about 69
- HTMLStripCharFilterFactory, type 69
- MappingCharFilterFactory, type 69
- PatternReplaceCharFilter, type 69

- ClassBridge** interface 66, 76
- clean goal** 24
- clear method** 101
- cluster**
 - applications, running in 118
 - master-slave clusters 119
 - simple cluster 118, 119
- components** 36
- conditional indexing**
 - about 76-78
 - EntityIndexingInterceptor interface, methods 78
- contextDestroyed method** 13
- contextInitialized method** 13
- COUNT_ASC option** 91
- COUNT_DESC option** 91
- createFacetingRequest method** 91
- createQuery method** 48
- currentDiscountPercentage member variable** 61
- currentDiscountPercentage property** 61

D

- date fields**
 - mapping 60
- default substring** 104
- deletes, individual updates** 101
- deviceName property** 83
- directory-based manager** 107
- DirectoryProvider**
 - about 109
 - Filesystem-based 109, 110
 - RAM-based 111
- discrete clause** 90
- documents** 29
- domain-specific language.** *See* DSL
- DSL** 16
- dynamic analyzer**
 - selecting 72-74
- dynamic boosting**
 - at index-time 75, 76

E

- elements** 36
- embedded objects** 36
- enableFullTextFilter method** 85

entities

- associated entities 32-34
- associated, querying 35, 36
- preparing, for Hibernate Search 10, 11
- relationship 32
- entity class**
 - creating 8-10
- EntityIndexingInterceptor interface, methods**
 - onAdd() method 78
 - onCollectionUpdate() method 78
 - onDelete() method 78
 - onUpdate() method 78
- EntityManager object** 45
- entity() method** 40
- execution mode, workers** 107

F

- Faceted search**
 - about 89-92
 - range facets 93, 94
- field**
 - multiple mapping, for same field 31
 - numeric fields, mapping 31, 32
- FieldBridge**
 - about 64
 - interface 66
 - multiple properties, combining into single field 66, 67
 - single variable, splitting into multiple fields 65
 - TwoWayFieldBridge 67
- Field.DEFAULT_NULL_TOKEN** 61
- field mapping**
 - options 30
- FIELD_VALUE option** 91
- Filesystem-based, DirectoryProvider** 109, 110
- filter definition**
 - creating 85
 - creating, for query 85
- filter factory**
 - creating 82, 83
- filtering**
 - about 81, 82
 - enabling, for query 85

- filter definition, creating 85
- filter factory, creating 82, 83
- filter key, adding 83, 84
- filter key**
 - adding 83, 84
- fire and forget mode** 107
- flushToIndexes method** 101
- forField element** 32
- Full Details button** 38, 86
- FullTextQuery class** 45
- FullTextQuery implementation** 45
- FullTextQuery object** 45, 48, 56, 81
- FullTextSession object** 15, 44, 47, 100, 102
- Fuzzy search** 48, 49

G

- getAnalyzerDefinitionName method** 73
- getFilter method** 82, 83
- getResultSize method** 56
- getSearchFactory method** 103
- getValue method** 92
- Groovy-based Gradle** 18

H

- H2**
 - URL 20
- hibernate-entitymanager dependency** 45
- Hibernate ORM**
 - about 88
 - API, selecting for 27, 28
- Hibernate ORM native API**
 - versus JPA entity mapping 28, 29
- HibernateQuery object** 16
- Hibernate Search**
 - application, running 21-24
 - build system, selecting 17, 18
 - entity class, creating 8-10
 - entity, preparing for 10, 11
 - filtering 81
 - importing 19, 20
 - incorporating, in application 7
 - projection 86
 - project, setting up for 19, 20 45, 46
 - search query code, writing 14-16
 - test data, loading 11, 13

- hibernate.search.default.directory_provider** 123
- hibernate.search.default.indexBase** 123
- hibernate.search.default.refresh** 123
- hibernate.search.default.sourceBase** 123
- hibernate.search.default.worker.backend** 123
- hibernate.search.default.worker.jms.connection_factory** 124
- hibernate.search.default.worker.jms.queue** 124
- Hibernate Search Domain-Specific Language.** *See* **Hibernate Search DSL**
- Hibernate Search DSL**
 - about 46
 - Boolean (combination) queries 53
 - Fuzzy search 48, 49
 - keyword query 47, 48
 - phrase query 50, 51
 - range query 52
 - Wildcard search 50
- HibernateSession class** 43
- HTMLStripCharFilterFactory, type** 69

I

- id field** 10
- if block** 86
- includeZeroCounts method** 91
- index** 30
- indexBase property** 110
- index.html page** 14
- IndexingOverride.APPLY_DEFAULT, enum values** 79
- IndexingOverride enum value** 78
- IndexingOverride.REMOVE, enum values** 78
- IndexingOverride.SKIP, enum values** 78
- IndexingOverride.UPDATE, enum values** 78
- index manager** 106, 107
- index method** 100
- indexNullAs** 30, 60
- individual updates**
 - about 100
 - add and update operation 100
 - and mass updates, differences 102

- deletes 101
- Infinispan index** 120
- interceptor** 76

J

- JavaDocs**
 - URL 83
- Java Persistence API.** *See* **JPA 2.0**
- javax.jms.Message object** 124
- JBoss Infinispan**
 - URL 120
- Jetty plugin**
 - adding, to Maven POM 21
- Jetty web server**
 - URL 21
- JGroups**
 - URL 120
- JMS message queue provider** 120
- JMS worker backend**
 - creating 123, 124
- JPA**
 - about 9
 - project, setting up for 45, 46
 - setting up, for Hibernate Search 45, 46
 - using, for queries 44, 45
- JPA 2.0** 9
- JPA entity mapping**
 - versus Hibernate ORM native API 28, 29

K

- keyword query** 47, 48

L

- LetterTokenizerFactory** 69
- locking strategy, DirectoryProvider** 110
- logical-AND operation** 53
- logical-OR operation** 54
- Lucene** 10, 11
- Lucene API** 83, 115
- luceneOptions.addFieldToDocument()** 65
- luceneOptions parameter** 65
- Luke**
 - Force unlock 113
 - URL 112
- Luke utility** 112-114

M

- manual indexing**
 - versus automatic indexing 99, 100
- manual optimization** 103
- mapping**
 - about 69
 - date fields 60
 - multiple mapping, for same field 31
 - numeric fields 31, 32
- mapping API**
 - versus query API 43, 44
- MappingCharFilterFactory, type** 69
- MassIndexer option** 101, 102
- mass updates**
 - about 101, 102
 - and individual updates, differences 102
- master node** 120
- MasterNodeInitializer class** 124
- master-slave clusters**
 - about 119
 - directory providers 120
 - example 121, 123
 - worker backends 120
- Maven**
 - characteristics 18
 - URL 18
- maven-archetype-webapp** 19
- maxFacetCount method** 91
- morning** 70
- multiple properties**
 - combining, into single field 66, 67
- must clause** 53
- mvn clean package** 26

N

- name** 30
- near-real-time** 106
- norms** 30
- null values**
 - handling 60

O

- object form**
 - projection results, converting to 87, 88
- objectToString method** 62, 67

- onAdd() method** 78
- onCollectionUpdate() method** 78
- onDelete() method** 78
- One-to-One custom conversion**
 - about 60
 - date fields, mapping 60
 - null values, handling 60, 61
- onField method** 51
- onUpdate() method** 78
- openSession(), public static synchronized method** 13
- optimization** 103
- optimizer strategy** 105, 106
- OptimizerStrategy interface** 105
- orderBy method** 91
- org.apache.lucene.search.Query object** 16
- org.hibernate.Query interface** 44
- org.hibernate.Query object** 16

P

- pagination** 56
- ParameterizedBridge** 63, 64
- partial indexing** 39, 40
- PatternReplaceCharFilter, type** 69
- PhoneticFilterFactory filter** 70
- phrase query** 50, 51, 83
- plain old Java object. *See* POJO**
- POJO** 9, 40
- production application**
 - performance improving, tips for 117, 118
- programmatic mapping API** 40-42
- project**
 - setting up, for Hibernate Search 45, 46
- projection**
 - about 86
 - based, query creating 87
 - Lucene fields, making available 88, 89
 - results, converting to object form 87, 88
- purgeAll operate** 101
- purge operate** 101

Q

- queries**
 - JPA, using 44, 45
- query**
 - time limits, placing 95, 96

- query API**
 - versus mapping API 43
- QueryBuilder class** 16, 47
- QueryBuilder object** 47
- query-time boosting** 95
- QueryTimeoutException exception** 96

R

- RAM-based** 111
- range query** 52
- README file** 18
- releaseDate field** 60
- Ruby-based Buildr** 18

S

- Scala-based SBT** 18
- SearchFactory class** 103
- SearchFullTextQuery object** 44
- search query**
 - code, writing 14-16
- searchString** 14
- segment** 103
- ServletContextListener** 11
- Session class** 44
- SessionFactory object** 13
- Session objects** 13
- setDeviceName setter method** 83
- setFirstResult method** 56
- setMaxResults method** 56
- setParameterValues method** 63
- setProjection method** 87
- sharding** 125-127
- single variable**
 - splitting, into multiple fields 65
- slave nodes** 120
- SnowballPorterFilterFactory filter** 70
- sorting** 54, 55
- StandardTokenizerFactory** 69
- startAndWait method** 102
- start method** 102
- StartupDataLoader class** 111, 122
- static analyzer**
 - selecting 71, 72
- static boosting**
 - at index-time 74, 75
- StopFilterFactory filter** 70

- store 30
- Store.COMPRESS 89
- Store.NO 89
- Store.YES 89
- StringBridge interface 61, 62
- string conversion
 - FieldBridge 64
 - ParameterizedBridge 63, 64
 - StringBridge 61, 62
 - TwoWayStringBridge 62
- stringToObject method 63
- supportedApps member variable 33

T

- testCluster() method 122
- test data
 - loading 11, 13
- thread pool, workers 108
- token filters
 - about 68, 69
 - PhoneticFilterFactory filter 70
 - SnowballPorterFilterFactory filter 70
 - StopFilterFactory filter 70
- tokenization 68
- tokenizer components
 - example 69
- transaction_limit.max 105
- tuples 87
- TwoWayFieldBridge 67
- TwoWayStringBridge 62

U

- useFileMappedBuffer parameter 22

V

- VAPORware Marketplace application 18
- VAPORware Marketplace web application 14
- very manual approach 104

W

- WhitespaceTokenizerFactory 69
- wildcard method 50
- wildcard search 50
- withSlop clause 51
- workers
 - about 107
 - buffer queue 108
 - execution mode 107
 - thread pool 108



Thank you for buying Hibernate Search by Example

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

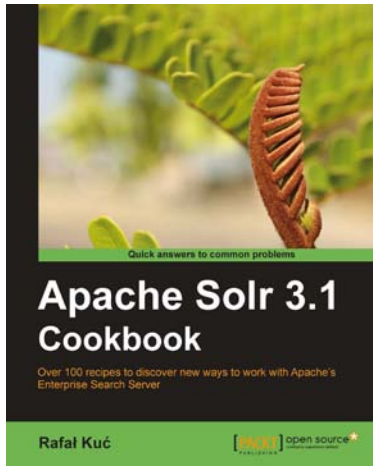
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



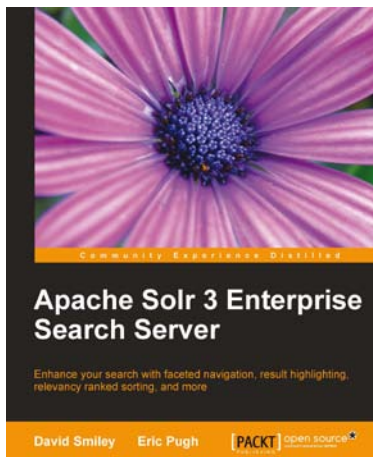
Apache Solr 3.1 Cookbook

ISBN: 978-1-84951-218-3

Paperback: 300 pages

Over 100 recipes to discover new ways to work with Apache's Enterprise Search Server

1. Improve the way in which you work with Apache Solr to make your search engine quicker and more effective
2. Deal with performance, setup, and configuration problems in no time
3. Discover little-known Solr functionalities and create your own modules to customize Solr to your company's needs



Apache Solr 3 Enterprise Search Server

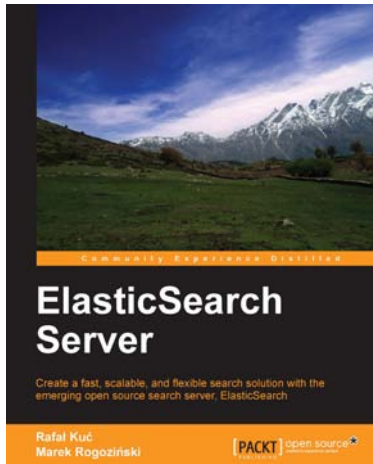
ISBN: 978-1-84951-606-8

Paperback: 418 pages

Enhance your search with faceted navigation, result highlighting, relevancy ranked sorting, and more

1. Comprehensive information on Apache Solr 3 with examples and tips so you can focus on the important parts
2. Integration examples with databases, web-crawlers, XSLT, Java & embedded-Solr, PHP & Drupal, JavaScript, Ruby frameworks
3. Advice on data modeling, deployment considerations to include security, logging, and monitoring, and advice on scaling Solr and measuring performance

Please check www.PacktPub.com for information on our titles

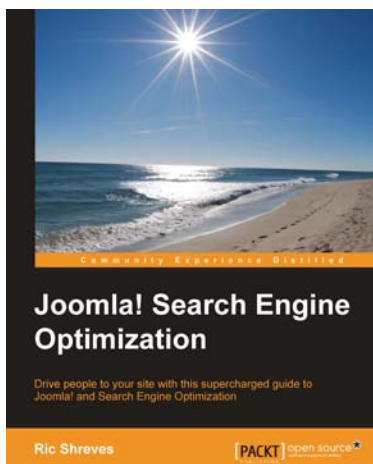


ElasticSearch Server

ISBN: 978-1-84951-844-4 Paperback: 318 pages

Create a fast, scalable, and flexible search solution with the emerging open source search server, ElasticSearch

1. Learn the basics of ElasticSearch like data indexing, analysis, and dynamic mapping
2. Query and filter ElasticSearch for more accurate and precise search results
3. Learn how to monitor and manage ElasticSearch clusters and troubleshoot any problems that arise



Joomla! Search Engine Optimization

ISBN: 978-1-84951-876-5 Paperback: 116 pages

Drive people to your site with this supercharged guide to Joomla! and Search Engine Optimization

1. Learn how to create a search engine-optimized Joomla! website
2. Packed full of tips to help you develop an appropriate SEO strategy
4. Discover the right configurations and extensions for SEO purposes

Please check **www.PacktPub.com** for information on our titles