Community Experience Distilled

Hibernate Search by Example

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

Steve Perkins

[PACKT] open source*
community experience distilled
PUBLISHING

Hibernate Search by Example

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

Steve Perkins

BIRMINGHAM - MUMBAI

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

Reviewers
Shaozhuang Liu
Murat Yener

Acquisition Editor
Joanne Fitzpatrick

Commissioning Editor
Meeta Rajani

Technical Editors
Amit Ramadas
Lubna Shaikh

Project Coordinator
Amigya Khurana

Proofreader
Ting Baker

Indexer
Monica Ajmera

Graphics
Sheetal Aute

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`.

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.

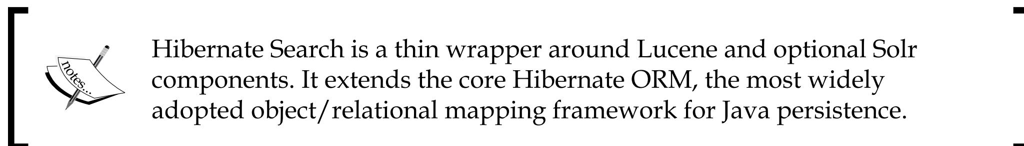
Above all else, a search should be *nice*. When we post something in an online forum and mistake the words "there", "they're", and "their", people might only criticize our grammar. By contrast, a search should simply understand our typos and be cool about it! A search is at its best when it pleasantly 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?

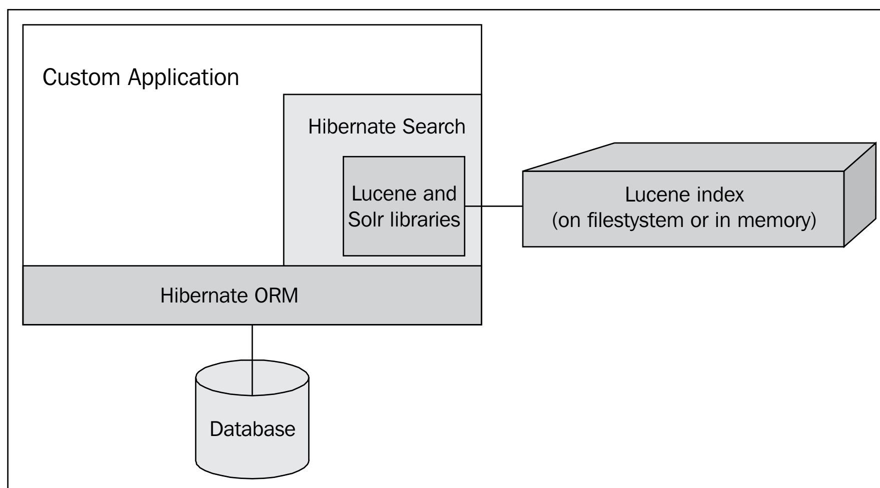
The true brain behind this search functionality is Apache Lucene, an open source software library for indexing and searching data. Lucene is an established Java project with a rich history of innovation, although it has been ported to other programming languages as well. It is widely adopted across a 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;
}
```

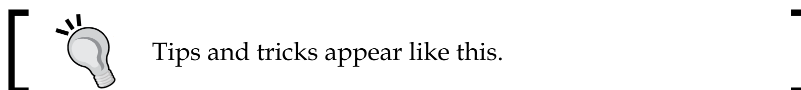
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

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

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

```
mvn archetype:generate -DgroupId=com.packpub.hibernatesearch.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 SessionFactorysessionFactory;

   /** Thread-safe helper method for creating Hibernate sessions. */
   public static synchronized Session openSession() {
      if(sessionFactory == null) {
         Configuration configuration = new Configuration();
         configuration.configure();
         ServiceRegistryserviceRegistry = 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 SearchServletextends 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();
FullTextSessionfullTextSession =
   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");

QueryBuilderqueryBuilder =
fullTextSession.getSearchFactory()
   .buildQueryBuilder().forEntity( App.class ).get();
org.apache.lucene.search.QueryluceneQuery =
   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"%>
<%@ tagliburi="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:forEachvar="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. `fieldsgroupId` 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>
```

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>
```