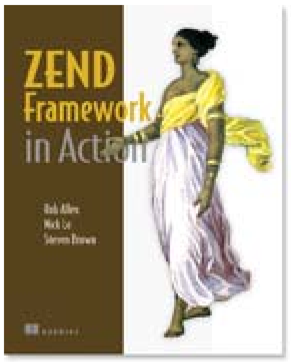
MANNING PUBLICATIONS

Introducing Zend_Search_Lucene

Excerpted from



Zend Framework in Action
EARLY ACCESS EDITION

Rob Allen, Nick Lo, Steven Brown

MEAP Release: September 2007
Softbound print: October 2008 (est.) | 425 pages
ISBN: 1933988320

*This article is based on chapter 9 from **Zend Framework in Action** by Rob Allen, Nick Lo, and Steven Brown.*

Zend Framework's search component, Zend_Search_Lucene is a very powerful tool. It is a full text search engine that is based on the popular Apache Lucene project which is a search engine for Java. The index files created by Zend_Search_Lucene are compatible with Apache Lucene and hence any of the index management utilities written for Apache Lucene will work with Zend_Search_Lucene too.

Zend_Search_Lucene creates an index of documents. The documents are each instances of Zend_Search_Lucene_Document. Each document contains Zend_Search_Lucene_Field objects which contain the actual data. A visual representation is shown in figure 1.

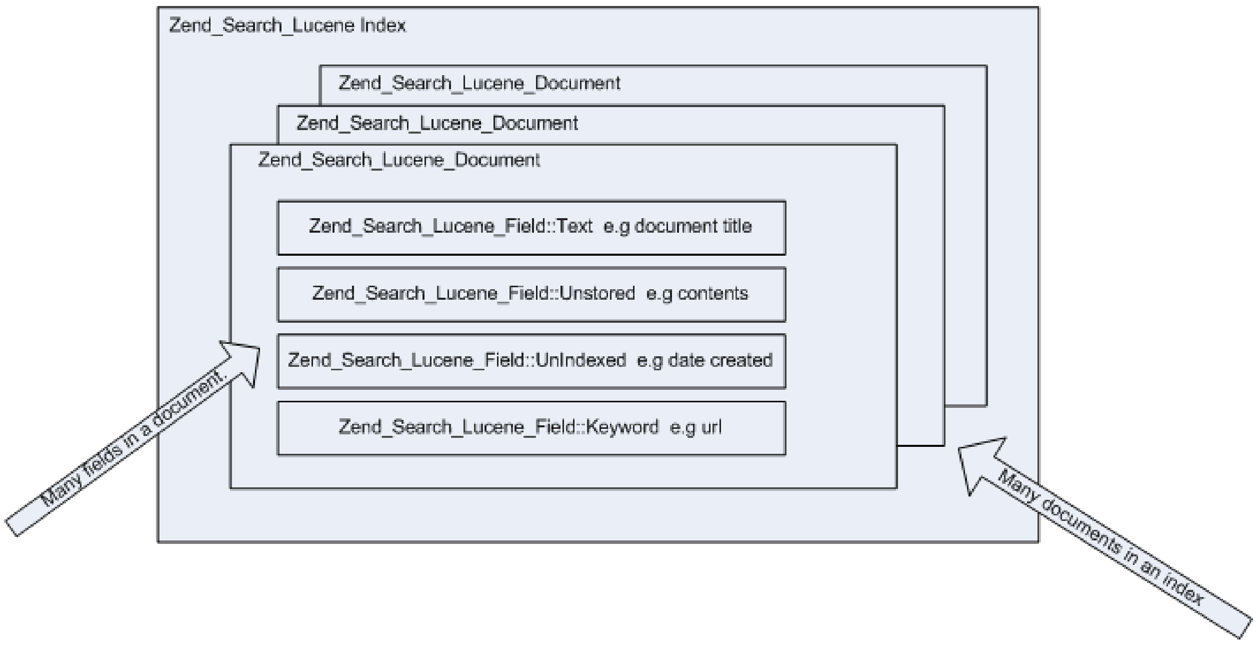

Zend_Search_Lucene Index
Zend_Search_Lucene_Document
Zend_Search_Lucene_Document
Zend_Search_Lucene_Document
Zend_Search_Lucene_Field::Text e.g document title
Zend_Search_Lucene_Field::Unstored e.g contents
Zend_Search_Lucene_Field::UnIndexed e.g date created
Zend_Search_Lucene_Field::Keyword e.g url
Many fields in a document.
Many documents in an index

Figure 1 A Zend_Search_Lucene index consists of multiple documents, each containing multiple fields. Not all fields are stored and some fields may contain data for display rather than searching.

Queries can then the issues against the index and an ordered array of results (each of type `Zend_Search_Lucene_Search_QueryHit`) is returned. The first part of implementing a solution using it is, therefore, to create an index.

A separate search index for your website

Creating an index for `Zend_Search_Lucene` is just a case of calling the `create()` function:

```
$index = Zend_Search_Lucene::create('path/to/index');
```

The index created is actually a directory that contains a few files in a format that is compatible with the main Apache Lucene project. This means that if you want to, you could create index files using the Java or .Net applications, or conversely, create indexes with Zend_Search_Lucene and search them using Java.

Having created the index, the next thing to do is to put data in it for searching on. This is where it gets complicated and we have to pay attention! Adding data is easily done using the `addDocument()` method, however you need to set up the fields within the document and each field has a type.

For example:

```
$doc = new Zend_Search_Lucene_Document();
$doc->addField(Zend_Search_Lucene_Field::UnIndexed('url', $url));
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents', $contents));
$doc->addField(Zend_Search_Lucene_Field::Text('desc', $desc));

$index->addDocument($doc);
```

As you can see in this example, the url data is of field type “UnIndexed”, contents are “UnStored” and the description is “Text”. Each field type is treated differently by the search engine to decide if it needs to store the actual data or just use it for creating the index file. Table 1 shows the key field types and what the differences are.

Table 1 Lucene field types for adding fields to an index

Name	Indexed	Stored	Tokenised	Notes
Keyword	Yes	Yes	No	Use for storing and indexing data that is already split into separate words for searching.
UnIndexed	No	Yes	No	Use for data that isn't searched on, but is displayed in the search results, such as dates or database id fields.
Text	Yes	Yes	Yes	Use for storing data that is both indexed and used in the search results display. This data is split into separate words for indexing.
Unstored	Yes	No	Yes	Use for indexing the main content of the document. The actual data isn't stored as you won't be using it for search results display.
Binary	No	Yes	No	Use for storing binary data that is associated with the document, such as a thumbnail.

There are two reasons for adding a field to the search index: providing the search data and providing for display of the results. The data fields available cover both these operations and choosing the correct field type for a given piece of information is crucial for the correct operation of the search engine. Let's look at storing the data for searching first. This is known as indexing and so the Keyword, Text and Unstored fields are relevant. The main difference between Keyword and Text/Unstored is the concept of tokenizing which is when the indexing engine needs to analyze the data to determine the actual words in use. The Keyword field is not tokenized which means that each word is used exactly as spelt for the purposes of search matching. The Text /Unstored fields however are tokenized, which means that each word is analyzed and the underlying base word is used for search matching. For example, punctuation and plurals are removed for each word.

When using a search engine, the list of results needs to provide enough information for the user to determine if a given result is the one she is looking for. The field types that are "Stored" can help with this as they are returned by the search engine. These are the Keyword, UnIndexed, Text and Binary field types. The Unindexed and Binary field types exist solely for storing data used in the results display. Typically, the Binary field type would be used for storing a thumbnail image that relates to the record and the UnIndexed field type is used to store items such as a summary of the result or data related to finding the result, such as its database table, id or URL.

`Zend_Search_Lucene` will automatically optimize the search index as new documents are added. You can also force optimization by calling the `optimize()` function if required though. Now that we have created an index file, we can now perform searches on the index. As you would expect, `Zend_Search_Lucene` provides a variety of powerful mechanisms for building queries that produce the desired results and we shall explore them next.

Powerful queries

Searching a `Zend_Search_Lucene` index is as simple as:

```
$index = Zend_Search_Lucene::open('path/to/index');
$index->find($query);
```

The $query parameter may be either a string or you can build a query using `Zend_Search_Lucene` objects and pass in an instance of `Zend_Search_Lucene_Search_Query`. Clearly passing in a string is the easiest, but for maximum flexibility, using the `Zend_Search_Lucene_Search_Query` object can be very useful. The string is converted to a search query object using a query parser and a good rule of thumb is that you should use the query parser for data from users and the query objects directly when programmatically creating queries. This implies that for an advanced search system that most websites provide, a hybrid approach would be used. We'll explore that a little later. Let's look first at the query parser for strings.

String queries

All search engines provide a very simple search interface for their users: a single text field. This makes it very easy to use, but at first glance seems to make it harder to provide a complex query. Like Google, Zend_Search_Lucene has a query parser that can interpret what is typed into a single text field into a powerful query. When you pass in a string to the find() function, behind the scenes the function Zend_Search_Lucene_Search_QueryParser::parse() is called. This class implements the Lucene query parser syntax as supported by v 2.0 of Apache Lucene.

To do its work, the parser breaks down the query into terms, phrases and operators. A term is a single word and a query is multiple words grouped using quotation marks, such as "hello world". An operator is a boolean word (such as AND) or symbol modified used to provide more complex queries. Wildcards are also supported using the asterisk and question mark symbols. A question mark is used to represent a single character and the asterisk represents several characters. For instance searching for frame* will find frame, framework, frameset and so on. Other

Table 2 lists the key modifier symbols that can be applied to a search term.

Table 2: Modifiers for search terms can be applied to control how the parser uses the term when searching.

Modifier	Symbol	Example	Description
Wildcards	? and *	H?t h*t	? is a single character placeholder and * represents multiple characters
Proximity	~x	"php power"~10	The terms must be within a certain number of words apart (10 in the example)
Inclusive Range	fieldname:[x TO y]	category:[skiing TO surfing]	Find all documents whose field values are between the upper and lower bounds specified
Exclusive Range	fieldname:{x To y}	published:[20070101 TO 20080101]	Find all documents whose field values are greater than the specified lower bound and less than the upper bound. This does not include the bounds.
Term Boost	^x	"Rob Allen"^3	Increase the relevance of a document containing this term. The number determines how much of a increase in relevance is given.

Each modifier in table 1 affects only the term to which it is attached. Operators on the other hand affect the makeup of the query. The key operators available are listed in Table 3.

Table 3: Boolean operators affect the makeup of the query and provide the power to refines the search

Operator	Symbol	Example	Description
Required	+	+php power	The term after the + symbol must exist in the document.
Prohibit	-		Documents containing the term after the – symbol are excluded from the search results.
AND	AND or &&	php and power	Both terms must exist anywhere in the document
OR	OR or ||	php or html	Either term must be present in all returned documents.
NOT	NOT or !	php not java	Only include documents that contain the first term but not the second term.

Careful use of the Boolean operators can create very complex queries however, in the “real world” most users are unlike to use more than one or two operators in any given query. The set of Boolean operators supported is pretty much the same as that used by major search engines, such as Google, and so your users will have a degree of familiarity.