Chapter No. 3
"Indexing Data"
In this package, you will find:
A Biography of the authors of the book
A preview chapter from the book, Chapter NO.3 "Indexing Data"
A synopsis of the book’s content
Information on where to buy this book

About the Authors
Born to code, **David Smiley** is a senior software developer and loves programming. He has 10 years of experience in the defense industry at MITRE, using Java and various web technologies. David is a strong believer in the opensource development model and has made small contributions to various projects over the years.

David began using Lucene way back in 2000 during its infancy and was immediately excited by it and its future potential. He later went on to use the Lucene based "Compass" library to construct a very basic search server, similar in spirit to Solr. Since then, David has used Solr in a major search project and was able to contribute modifications back to the Solr community. Although preferring open source solutions, David has also been trained on the commercial Endeca search platform and is currently using that product as well as Solr for different projects.

For More Information:
Most, if not all, authors seem to dedicate their book to someone. As simply a reader of books, I have thought of this seeming prerequisite as customary tradition. That was my feeling before I embarked on writing about Solr, a project that has sapped my previously "free" time on nights and weekends for a year. I chose this sacrifice and would not change it, but my wife, family, and friends did not choose it. I am married to my lovely wife Sylvie who has sacrificed easily as much as I have to complete this book. She has suffered through this time with an absentee husband while bearing our first child—Camille. She was born about a week before the completion of my first draft and has been the apple of my eye ever since. I officially dedicate this book to my wife Sylvie and my daughter Camille, whom I both lovingly adore. I also pledge to read book dedications with newfound firsthand experience at what the dedication represents.

I would also like to thank others who helped bring this book to fruition. Namely, if it were not for Doug Cutting creating Lucene with an open source license, there would be no Solr. Furthermore, CNet's decision to open source what was an in-house project, Solr itself in 2006, deserves praise. Many corporations do not understand that open source isn't just "free code" you get for free that others wrote; it is an opportunity to let your code flourish on the outside instead of it withering inside. Finally, I thank the team at Packt who were particularly patient with me as a first-time author writing at a pace that left a lot to be desired.

Last but not least, this book would not have been completed in a reasonable time were it not for the assistance of my contributing author, Eric Pugh. His perspectives and experiences have complemented mine so well that I am absolutely certain the quality of this book is much better than what I could have done alone.

Thank you all.

Fascinated by the 'craft' of software development, Eric Pugh has been heavily involved in the open source world as a developer, committer, and user for the past five years. He is an emeritus member of the Apache Software Foundation and lately has been mulling over how we move from the read/write Web to the read/write/share Web.

In biotech, financial services, and defense IT, he has helped European and American companies develop coherent strategies for embracing open source software. As a speaker, he has advocated the advantages of Agile practices in software development.
Eric became involved with Solr when he submitted the patch SOLR-284 for Parsing Rich Document types such as PDF and MS Office formats that became the single most popular patch as measured by votes! The patch was subsequently cleaned up and enhanced by three other individuals, demonstrating the power of the open source model to build great code collaboratively. SOLR-284 was eventually refactored into Solr Cell as part of Solr version 1.4.

He blogs at http://www.opensourceconnections.com/blog/.

Throughout my life I have been helped by so many people, but all too rarely do I get to explicitly thank them. This book is arguably one of the high points of my career, and as I wrote it, I thought about all the people who have provided encouragement, mentoring, and the occasional push to succeed. First off, I would like to thank Erik Hatcher, author, entrepreneur, and great family man for introducing me to the world of open source software. My first hesitant patch to Ant was made under his tutelage, and later my interest in Solr was fanned by his advocacy. Thanks to Harry Sleeper for taking a chance on a first time conference speaker; he moved me from thinking of myself as a developer improving myself to thinking of myself as a consultant improving the world (of software!). His team at MITRE are some of the most passionate developers I have met, and it was through them I met my co-author David. I owe a huge debt of gratitude to David Smiley. He has encouraged me, coached me, and put up with my lack of respect for book deadlines, making this book project a very positive experience! I look forward to the next one. With my new son Morgan at home, I could only have done this project with a generous support of time from my company, OpenSource Connections. I am incredibly proud of what o19s is accomplishing!

Lastly, to the all the folks in the Solr/Lucene community who took the time to review early drafts and provide feedback: Solr is at the tipping point of becoming the "it" search engine because of your passion and commitment

I am who I am because of my wife, Kate. Schweeatie, real life for me began when we met. Thank you.
Solr 1.4 Enterprise Search Server

Text search has been around for perhaps longer than we all can remember. Just about all systems, from client installed software to web sites to the web itself, have search. Yet there is a big difference between the best search experiences and the mediocre, unmemorable ones. If you want the application you're building to stand out above the rest, then it's got to have great search features. If you leave this to the capabilities of a database, then it's near impossible that you're going to get a great search experience, because it's not going to have features that users come to expect in a great search. With Solr, the leading open source search server, you'll tap into a host of features from highlighting search results to spell-checking to faceting.

As you read Solr Enterprise Search Server you'll be guided through all of the aspects of Solr, from the initial download to eventual deployment and performance optimization. Nearly all the options of Solr are listed and described here, thus making this book a resource to turn to as you implement your Solr based solution. The book contains code examples in several programming languages that explore various integration options, such as implementing query auto-complete in a web browser and integrating a web crawler. You'll find these working examples in the online supplement to the book along with a large, real-world, openly available data set from MusicBrainz.org. Furthermore, you will also find instructions on accessing a Solr image readily deployed from within Amazon's Elastic Compute Cloud.

Solr Enterprise Search Server targets the Solr 1.4 version. However, as this book went to print prior to Solr 1.4’s release, two features were not incorporated into the book: search result clustering and trie-range numeric fields.
What This Book Covers

Chapter 1, Quick Starting Solr introduces Solr to the reader as a middle ground between database technology and document/web crawlers. The reader is guided through the Solr distribution including running the sample configuration with sample data.

Chapter 2, The Schema and Text Analysis is all about Solr's schema. The schema design is an important first order of business along with the related text analysis configuration.

Chapter 3, Indexing Data details several methods to import data; most of them can be used to bring the MusicBrainz data set into the index. A popular Solr extension called the DataImportHandler is demonstrated too.

Chapter 4, Basic Searching is a thorough reference to Solr's query syntax from the basics to range queries. Factors influencing Solr's scoring algorithm are explained here, as well as diagnostic output essential to understanding how the query worked and how a score is computed.

Chapter 5, Enhanced Searching moves on to more querying topics. Various score boosting methods are explained from those based on record-level data to those that match particular fields or those that contain certain words. Next, faceting is a major subject area of this chapter. Finally, the term auto-complete is demonstrated, which is implemented by the faceting mechanism.

Chapter 6, Search Components covers a variety of searching extras in the form of Solr "components", namely, spell-check suggestions, highlighting search results, computing statistics of numeric fields, editorial alterations to specific user queries, and finding other records "more like this".

Chapter 7, Deployment transits from running Solr from a developer-centric perspective to deploying and running Solr as a deployed production enterprise service that is secure, has robust logging, and can be managed by System Administrators.

Chapter 8, Integrating Solr surveys a plethora of integration options for Solr, from supported client libraries in Java, JavaScript, and Ruby, to being able to consume Solr results in XML, JSON, and even PHP syntaxes. We’ll look at some best practices and approaches for integrating Solr into your web application.

Chapter 9, Scaling Solr looks at how to scale Solr up and out to avoid meltdown and meet performance expectations. This information varies from small changes of configuration files to architectural options.
Indexing Data

With a first cut of the schema defined, it's time to get data into the index. In this chapter, we're going to review the four main mechanisms that Solr offers:

- Solr's native XML
- CSV (Character Separated Value)
- Direct Database and XML Import through Solr's DataImportHandler
- Rich documents through Solr Cell

You will also find some options in Chapter 8 that have to do with language bindings and framework integration. All of them generally use Solr's native XML format, which we'll get to right away.

Communicating with Solr

There are a few dimensions to the options available for communicating with Solr:

Direct HTTP or a convenient client API

Applications interact with Solr over HTTP. This can either be done directly (by hand, but by using an HTTP client of your choice), or it might be facilitated by a Solr integration API such as SolrJ or Solr Flare, which in turn use HTTP. Such APIs are discussed in Chapter 8.

An exception to HTTP is offered by SolrJ, which can optionally be used in an embedded fashion with Solr (so-called Embedded Solr) to avoid network and interprocess communication altogether. However, unless you are sure you really want to embed Solr within another application, this option is discouraged in favor of writing a custom Solr updating request handler. More information about SolrJ and EmbeddedSolr is in Chapter 8.
Indexing Data

Data streamed remotely or from Solr's filesystem

Even though an application will be communicating with Solr over HTTP, it does not have to send Solr data over this channel. Solr supports what it calls remote streaming. Instead of giving Solr the data directly, it is given a URL that it will resolve. It might be an HTTP URL, but more likely it is a filesystem based URL, applicable when the data is already on Solr's machine. Finally, in the case of Solr's DataImportHandler, the data can be fetched from a database.

Data formats

The following are the different data formats:

- **Solr-XML**: Solr has a specific XML schema it uses to specify documents and their fields. It supports instructions to delete documents and to perform optimizes and commits too.
- **Solr-binary**: Analogous to Solr-XML, it is an efficient binary representation of the same structure. This is only supported by the SolrJ client API.
- **CSV**: CSV is a character separated value format (often a comma).
- **Rich documents like PDF, XLS, DOC, PPT to Solr**: The text data extracted from these formats is directed to a particular field in your Solr schema.
- **Finally, Solr's DIH DataImportHandler contrib add-on is a powerful capability that can communicate with both databases and XML sources (for example: web services). It supports configurable relational and schema mapping options and supports custom transformation additions if needed. The DIH uniquely supports delta updates if the source data has modification dates.**

We'll use the XML, CSV, and DIH options in bringing the MusicBrainz data into Solr from its database to demonstrate Solr's capability. Most likely, an application would use just one format.

Before these approaches are described, we'll discuss curl and remote streaming, which are foundational topics.

Using curl to interact with Solr

Solr receives commands (and possibly the associated data) through HTTP POST.
Solr lets you use HTTP GET too (for example, through your web browser). However, this is an inappropriate HTTP verb if it causes something to change on the server, as happens with indexing. For more information on this concept, read about REST at [http://en.wikipedia.org/wiki/Representational_State_Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer).

One way to send an HTTP POST is through the Unix command line program `curl` (also available on Windows through Cygwin). Even if you don't use `curl`, it is very important to know how we're going to use it, because the concepts will be applied no matter how you make the HTTP messages.

There are several ways to tell Solr to index data, and all of them are through HTTP POST:

- Send the data as the entire POST payload (only applicable to Solr's XML format). `curl` does this with `data-binary` (or some similar options) and an appropriate content-type header reflecting that it's XML.
- Send some name-value pairs akin to an HTML form submission. With `curl`, such pairs are proceeded by `-F`. If you're giving data to Solr to be indexed (as opposed to it looking for it in a database), then there are a few ways to do that:
  - Put the data into the `stream.body` parameter. If it's small, perhaps less than a megabyte, then this approach is fine. The limit is configured with the `multipartUploadLimitInKB` setting in `solrconfig.xml`.
  - Refer to the data through either a local file on the Solr server using the `stream.file` parameter or a URL that Solr will fetch it from through the `stream.url` parameter. These choices are a feature that Solr calls remote streaming.

Here is an example of the first choice. Let's say we have an XML file named `artists.xml` in the current directory. We can post it to Solr using the following command line:

```
curl http://localhost:8983/solr/update -H 'Content-type:text/xml; charset=utf-8' --data-binary @artists.xml
```

If it succeeds, then you'll have output that looks like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int><int name="QTime">128</int>
  </lst>
</response>
```
Indexing Data

To use the solr.body feature for the example above, you would do this:

curl http://localhost:8983/solr/update -F solr.body=@artists.xml

In both cases, the @ character instructs curl to get the data from the file instead of being @artists.xml literally. If the XML is short, then you can just as easily specify it literally on the command line:

curl http://localhost:8983/solr/update -F stream.body=' <commit />'

Notice the leading space in the value. This was intentional. In this example, curl treats @ and < to mean things we don't want. In this case, it might be more appropriate to use form-string instead of -F. However, it's more typing, and I'm feeling lazy.

Remote streaming

In the examples above, we've given Solr the data to index in the HTTP message. Alternatively, the POST request can give Solr a pointer to the data in the form of either a file path accessible to Solr or an HTTP URL to it.

The file path is accessed by the Solr server on its machine, not the client, and it must also have the necessary operating system file permissions too.

However, just as before, the originating request does not return a response until Solr has finished processing it. If you're sending a large CSV file, then it is practical to use remote streaming. Otherwise, if the file is of a decent size or is already at some known URL, then you may find remote streaming faster and/or more convenient, depending on your situation.

Here is an example of Solr accessing a local file:


To use a URL, the parameter would change to stream.url, and we'd specify a URL. We're passing a name-value parameter (stream.file and the path), not the actual data.
Remote streaming must be enabled

In order to use remote streaming \(\text{stream.file}\) or \(\text{stream.url}\), you must enable it in \text{solrconfig.xml}. It is disabled by default and is configured on a line that looks like this:

\[
\begin{align*}
\text{<requestParsers enableRemoteStreaming="true" multipartUploadLimitInKB="2048" />}
\end{align*}
\]

Sending XML to Solr

Solr's native XML syntax is very simple. You can tell Solr to add documents to an index, to commit changes, to optimize the index, and to delete documents. Here is a sample XML file you can HTTP POST to Solr:

\[
\begin{align*}
\text{<add allowDups="false">}
\text{<doc boost="2.0">}
\text{<field name="id">5432a</field>}
\text{<field name="type" ...</field>}
\text{<field name="a_name" boost="0.5"></field>}
\text{<!-- the date/time syntax MUST look just like this (ISO-8601)-->
\text{<field name="begin_date">2007-12-31T09:40:00Z</field>}
\text{</doc>}
\text{<doc>}
\text{<field name="id">5432a</field>}
\text{<field name="type" ...}
\text{<field name="begin_date">2007-12-31T09:40:00Z</field>}
\text{</doc>}
\text{<!-- more here as needed -->}
\text{</add>}
\end{align*}
\]

The \text{allowDups} defaults to \text{false} to guarantee the uniqueness of values in the field that you have designated as the unique field in the schema (assuming you have such a field). If you were to add another document that has the same value for the unique field, then this document would override the previous document, whether it is pending a commit or it's already committed. You will not get an error.

If you are sure that you will be adding a document that is not a duplicate, then you can set \text{allowDups} to \text{true} to get a performance improvement.
Boosting affects the scores of matching documents in order to affect ranking in score-sorted search results. Providing a boost value, whether at the document or field level, is optional. The default value is 1.0, which is effectively a non-boost. Technically, documents are not boosted, only fields are. The effective boost value of a field is that specified for the document multiplied by that specified for the field.

Specifying boosts here is called index-time boosting, which is rarely done as compared to the more flexible query-time boosting. Index-time boosting is less flexible because such boosting decisions must be decided at index-time and will apply to all of the queries.

Deleting documents
You can delete a document by its unique field (we delete two documents here):

    <delete><id>artist:11604</id><id>artist:11603</id></delete>

Or, you can delete all of the documents that match a particular Lucene/Solr query (the query syntax is not discussed in this chapter):

    <delete><query>timestamp: [* TO NOW-12HOUR]</query></delete>

The contents of the delete tag can be any number of ID and query tags if you want to batch many deletions into one message to Solr.

The query syntax is not discussed in this chapter, but I'll explain this somewhat complicated query anyway. Let's suppose that all of your documents had a timestamp field with a value of the time it was indexed, and you have an update strategy that bulk loads all of the data on a daily basis. If the loading process results in documents that shouldn't be in the index anymore, then we can delete them immediately after a bulk load. This query would delete all of the documents not indexed within the last 12 hours. Twelve was chosen somewhat arbitrarily, but it needs to be less than 24 (the update process interval) and greater than the longest time it might conceivably take to bulk load all the data.

Commit, optimize, and rollback
Data sent to Solr is not immediately searchable, nor do deletions take immediate effect. Like a database, changes must be committed first. Unlike a database, there are no distinct sessions (that is transactions) between each client, and instead there is in-effect one global modification state. This means that if more than one Solr client were to submit modifications and commit them at similar times, it is possible for part of one client's set of changes to be committed before that client told Solr to commit. Usually, you will have just one process responsible for updating Solr. But if not, then keep this in mind.
To commit changes using the XML syntax, simply send this to Solr:

```xml
<commit />
```

Depending on Solr's auto-warming configuration and cache state prior to committing (a Chapter 9 topic), a commit can take a non-trivial amount of time, in the order of seconds, perhaps up to a minute or longer in extreme cases. The amount of data committed has little impact on this delay. Generally, databases commit very fast. The last chapter deals with performance.

All uncommitted changes can be withdrawn by sending Solr the `rollback` command:

```xml
<rollback />
```

Lucene's index is internally composed of one or more segments. Modifications get committed to the last segment. Lucene will on occasions either start a new segment or merge them all together into one. When Lucene has just one segment, it is in an optimized state, because each segment degrades query performance. It is recommended to explicitly optimize the index at an opportune time like after a bulk load of data and/or a daily interval in off-peak hours, if there are sporadic updates to the index. You can do this by simply sending this XML:

```xml
<optimize />
```

Both `commit` and `optimize` take two additional boolean options that default to true:

```xml
<commit waitFlush="true" waitSearcher="true">
```

If you were to set these to false, then commit and optimize return immediately, even though the operation hasn't actually finished yet. So if you wrote a script that committed with these at their false values and then executed a query against Solr, then you may find that the query will not reflect the changes (yet). By waiting for the data to flush to the disk (waitFlush) and waiting for a new searcher to be ready to respond to changes (waitSearcher), this circumstance is avoided.

A convenient alternative to send these commands through XML is to simply add `commit`, `optimize`, or `rollback` as boolean request parameters when communicating with Solr. You'll see an example of this with CSV in the next section. Request parameters can be put on the URL and/or as form parameters, if applicable. These three request parameters are honored by Solr whether you send Solr its native XML format, CSV, or rich documents. `waitFlush` and `waitSearcher` are not supported in this manner.
Sending CSV to Solr

If you have data in a CSV format or if it is more convenient for you to get CSV than XML, then you may prefer the CSV option to the XML format. Solr's CSV options are fairly flexible.

To get some CSV data out of a local PostgreSQL database for the MusicBrainz tracks, I ran this command:

```
psql -U postgres -d musicbrainz_db -c "COPY (" \n  select 'Track:' || t.id as id, 'Track' as type, t.name as t_name, 
  t.length/1000 as t_duration, a.id as t_a_id, a.name as t_a_name, 
  albumjoin.sequence as t_num, r.id as t_r_id, r.name as t_r_name, array_to_string(r.attributes,' ') as t_r_attributes, albummeta.tracks as t_r_tracks 
  from (track t inner join albumjoin on t.id = albumjoin.track 
    inner join album r on albumjoin.album = r.id left join albummeta on albumjoin.album = albummeta.id) inner join artist a on t.artist = a.id \n) to '/tmp/tracks' CSV HEADER"
```

And it generated output that looks like this (first three lines):

```
id,type,t_name,t_duration,t_a_id,t_a_name,t_num,t_r_id,t_r_name,t_r_attributes,t_r_tracks
Track:183326,Track,In the Arms of Sleep,254,11650,The Smashing Pumpkins,4,22471,Mellon Collie and the Infinite Sadness (disc 2: Twilight to Starlight),0 1 100,14
Track:183328,Track,Tales of a Scorched Earth,228,11650,The Smashing Pumpkins,6,22471,Mellon Collie and the Infinite Sadness (disc 2: Twilight to Starlight),0 1 100,14
```

To get Solr to import the CSV file, I typed this at the command line:

```
curl http://localhost:8983/solr/update/csv -F f.t_r_attributes.split=true -F f.t_r_attributes.separator=' ' -F overwrite=false -F commit=true -F stream.file=/tmp/tracks
```

When I actually did this I had PostgreSQL on one machine and Solr on another. I used the Unix `mkfifo` command to create an in-memory data pipe mounted at `/tmp/tracks`. This way, I didn't have to actually generate a huge CSV file. I could essentially stream it directly from PostgreSQL into Solr. Details on this approach and PostgreSQL are out of the scope of this book.
Configuration options
The configuration options to Solr’s CSV capability are set through HTTP posting name-value pairs in the same format that HTML forms post their data. As explained earlier, technically you could use a URL through HTTP GET with a stream.url or stream.file parameter. However, this is a bad practice. Also note that Solr’s CSV capability doesn't support index-time boosting, but that is an uncommon requirement.

The following are the names of each configuration option with an explanation. For the MusicBrainz track CSV file, I was able to use the defaults with the exception of specifying how to parse the multi-valued t_r_attributes field and disabling unique key processing for performance.

- **separator**: The character that separates each value on a line. Defaults to ,.
- **header**: Is set to true if the first line lists the field names (the default).
- **fieldnames**: If the first line doesn't have the field names, then you’ll have to use this instead to indicate what they are. They are comma separated. If no name is specified for a column, then its data is skipped.
- **skip**: The fields to not import in the CSV file.
- **skipLines**: The number of lines to skip in the input file. Defaults to 0.
- **trim**: If true, then removes leading and trailing whitespace as a final step, even if quoting is used to explicitly specify a space. Defaults to false. Solr already does an initial pass trim, but quoting may leave spaces.
- **encapsulator**: This character is used to encapsulate (that is surround, quote) values in order to preserve the field separator as a field value instead of mistakenly parsing it as the next field. This character itself is escaped by doubling it. It defaults to the double quote, unless escape is specified. Example:

  11604, foo, "The ""second"" word is quoted.", bar

- **escape**: If this character is found in the input text, then the next character is taken literally in place of this escape character, and it isn't otherwise treated specially by the file's syntax. Example:

  11604, foo, The second\, word is followed by a comma., bar

- **keepEmpty**: Specified whether blank (zero length) fields should be indexed as such or omitted. It defaults to false.
- **overwrite**: It indicates whether to enforce the unique key constraint of the schema by overwriting existing documents with the same ID. It defaults to true. Disable this to increase performance, if you are sure you are passing new documents.
Indexing Data

- **split**: This is a field-specific option used to split what would normally be one value into multiple values. Another set of CSV configuration options (separator, and so on) can be specified for this field to instruct Solr on how to do that. See the previous tracks MusicBrainz example on how this is used.

- **map**: This is another field-specific option used to replace input values with another. It can be used to remove values too. The value should include a colon which separates the left side which is replaced with the right side. If we were to use this feature on the tracks of the MusicBrainz data, then it could be used to map the numeric code in \texttt{t\_r\_attributes} to more meaningful values. Here's an example of such an attempt:

  ```
  -F keepEmpty=false -F f.t\_r\_attributes.map=0:
  -F f.t\_r\_attributes.map=1:Album -F f.t\_r\_attributes.map=2:Single
  ```

  This causes 0 to be removed, because it seems to be useless data, as nearly all tracks have it, and we map 1 to Album and 2 to Single.

### Direct database and XML import

The capability for Solr to get data directly from a database or HTTP GET accessible XML is distributed with Solr as a contrib module, and it is known as the DataImportHandler (DIH in short). The complete reference documentation for this capability is here at http://wiki.apache.org/solr/DataImportHandler, and it's rather thorough. In this chapter, we'll only walk through an example to see how it can be used with the MusicBrainz data set.

In short, the DIH offers the following capabilities:

- Imports data from databases through **JDBC** (Java Database Connectivity)
- Imports XML data from a URL (HTTP GET) or a file
- Can combine data from different tables or sources in various ways
- Extraction/Transformation of the data
- Import of updated (delta) data from a database, assuming a last-updated date
- A diagnostic/development web page
- Extensible to support alternative data sources and transformation steps

As the MusicBrainz data is in a database, the most direct method to get data into Solr is definitely through the DIH using JDBC.
Getting started with DIH

DIH is not a direct part of Solr. Hence it might not be included in your Solr setup. It amounts to a JAR file named something like apache-solr-dataimporthandler-1.4.jar, which is probably already embedded within the solr.war file. You can use `jar -tf solr.war` to see. Alternatively, it may be placed in `<solr-home>/lib`, which is alongside the conf directory we've been working with. For database connectivity, we need to ensure that the JDBC driver is on the Java classpath. Placing it in `<solr-home>/lib` is a convenient way to do this.

The DIH needs to be registered with Solr in solrconfig.xml. Here is how it is done:

```xml
<requestHandler name="/dataimport"
    class="org.apache.solr.handler.dataimport.DataImportHandler">
    <lst name="defaults">
        <str name="config">mb-dih-artists-jdbc.xml</str>
    </lst>
</requestHandler>
```

`mb-dih-artists-jdbc.xml` is a file in `<solr-home>/conf`, which is used to configure DIH. It is possible to specify some configuration aspects in this request handler configuration instead of the dedicated configuration file. However, I recommend that it all be in the DIH config file, as in our example here.

Given below is an `mb-dih-artists-jdbc.xml` file with a rather long SQL query:

```xml
<dataConfig>
    <dataSource name="jdbc" driver="org.postgresql.Driver"
        url="jdbc:postgresql://localhost/musicbrainz_db"
        user="musicbrainz" readOnly="true" autoCommit="false" />
    <document>
        <entity name="artist" dataSource="jdbc" pk="id" query="select
            a.id as id,
            a.name as a_name, a.sortname as a_name_sort,
            a.begindate as a_begin_date, a.enddate as a_end_date,
            a.type as a_type
        ,array_to_string(
            array(select aa.name from artistalias aa
            where aa.ref = a.id )
        , '|') as a_alias
        ,array_to_string(
```
The DIH development console

Before describing the configuration details, we're going to take a look at the DIH development console. It is accessed by going to this URL (modifications may be needed for your host, port, core, and so on):

http://localhost:8983/solr/admin/dataimport.jsp
The development console looks like the following screenshot:

The screen is divided into two panes: on the left is the DIH control form, which includes an editable version of the DIH configuration file and on the right is the command output as raw XML. The screen works quite simply. The form essentially results in submitting a URL to the right pane. There’s no real server-side logic to this interface beyond the standard DIH command invocations being executed on the right. The last section on DIH in this chapter goes into more detail on submitting a command to the DIH.

**DIH DataSources of type JdbcDataSource**

The DIH configuration file starts with the declaration of one or more data sources using the element `<dataSource/>`, which refers to either a database, a file, or an HTTP URL, depending on the `type` attribute. It defaults to a value of `JdbcDataSource`. Those familiar with JDBC should find the `driver` and `url` attributes with accompanying `user` and `password` straightforward—consult the documentation for your driver/database for further information. `readOnly` is a boolean that will set a variety of other JDBC options appropriately when set.
Indexing Data

to true. And **batchSize** is an alias for the JDBC **fetchSize** and defaults to 500. There are numerous JDBC oriented attributes that can be set as well. I would not normally recommend learning about a feature by reading source code, but this is an exception. For further information, read [org.apache.solr.handler.dataimport.JdbcDataSource.java](http://www.packtpub.com/solr-1-4-enterprise-search-server/book).

### Efficient JDBC configuration

Many database drivers in the default configurations (including those for PostgreSQL and MySQL) fetch all of the query results into the memory instead of on-demand or using a batch/fetch size. This may work well for typical database usage like **OLTP (Online Transaction Processing systems)**, but is completely unworkable for **ETL (Extract Transform and Load)** usage such as this. Configuring the driver to stream the data requires driver-specific configuration options. You may need to consult relevant documentation for the JDBC driver. For PostgreSQL, set **autoCommit** to false. For MySQL, set **batchSize** to -1 (The DIH detects the -1 and replaces it with Java’s **Integer.MIN_VALUE**, which triggers the special behavior in MySQL’s JDBC driver). For Microsoft SQL Server, set **responseBuffering** to **adaptive**. Further information about specific databases is at [http://wiki.apache.org/solr/DataImportHandlerFaq](http://www.packtpub.com/solr-1-4-enterprise-search-server/book).

### DIH documents, entities

After the declaration of `<dataSource/>` element(s) is the `<document/>` element. In turn, this element contains one or more `<entity/>` elements. In this sample configuration, we’re only getting artists. However, if we wanted to have more than one type in the same index, then another could be added. The **dataSource** attribute references a correspondingly named element earlier. It is only necessary if there are multiple to choose from, but we've put it here explicitly anyway.

The main piece of an entity used with a JDBC data source is the **query** attribute, which is the SQL query to be evaluated. You’ll notice that this query involves some sub-queries, which are made into arrays and then transformed into strings joined by spaces. The particular functions used to do these sorts of things are generally database specific. This is done to shoe-horn multi-valued data into a single row in the results. It may create a more complicated query, but it does mean that the database does all of the heavy lifting so that all of the data Solr needs for an artist is in the row. An alternative with DIH is to declare other entities within the entity. If you aren’t using a database or if you wish to mix in another data source (even if it’s of a different type), then you will be forced to do that. See the Solr DIH Wiki page for examples: [http://wiki.apache.org/solr/DataImportHandler](http://www.packtpub.com/solr-1-4-enterprise-search-server/book).
The DIH also supports a delta query, which is a query that selects time-stamped data with dates after the last queried date. This won't be covered here, but you can find more information at the previous URL.

**DIH fields and transformers**

Within the `<entity/>` are some `<field/>` elements that declare how the columns in the query map to Solr. The field element must have a `column` attribute that matches the corresponding named column in the SQL query. The `name` attribute is the Solr schema field name that the column is going into. If it is not specified (and it never is for our example), then it defaults to the column name.

Use the SQL as a keyword as we've done to use the same names as the Solr schema instead of the database schema. This reduces the number of explicit mappings needed in `<field/>` elements and shortens existing ones.

When a column in the result can be placed directly into Solr without further processing, there is no need to specify the field declaration, because it is implied.

An attribute of the entity declaration that we didn't mention yet is `transformer`. This declares a comma-separated list of transformers that manipulate the transfer of data from the JDBC resultset into a Solr field. These transformers evaluate a field, if it has an attribute it uses to do its job. More than one might operate on a given field. Therefore, the order in which the transformers are declared in matters. Here are the attributes we've used:

- **template**: It is used by TemplateTransformer and declares text, which might include variable name substitutions using `${name}` syntax. To access an existing field, use the `entityname.columnname` syntax.

- **splitBy**: It is used by RegexTransformer and splits a single string value into a multi-value by looking for the specified character.

- **dateTimeFormat**: It is used by DateFormatTransformer. This is a Java date/time format pattern ([http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html](http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html)). If the type of the field in the schema is a date, then it is necessary to ensure Solr can interpret the format. Alternatively, ensure that the string matches the ISO-8601 format, which looks like this: `1976-10-23T23:59:59.000Z`. As in all cases in Solr, when specifying dates you can use its so-called "DateMath" syntax (described in the next chapter) such as appending `/DAY` to tell Solr to round the date to a day.
Indexing Data

Importing with DIH

Unlike the other importing mechanisms, the DIH returns immediately, while the import continues asynchronously. To get the current status of the DIH, go to this URL `http://localhost:8983/solr/dataimport`, and you'll get output like the following:

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">15</int>
  </lst>
  <lst name="initArgs">
    <lst name="defaults">
      <str name="config">mb-dih-artists-jdbc.xml</str>
    </lst>
  </lst>
  <str name="status">idle</str>
  <str name="importResponse"/>
  <lst name="statusMessages"/>
  <str name="WARNING">This response format is experimental. It is likely to change in the future.</str>
</response>
```

Commands are given to DIH as request parameters just as everything else is in Solr. We could tell the DIH to do a full-import just by going to this URL: `http://localhost:8983/solr/dataimport?command=full-import`. On the command line we would use:

```
curl http://localhost:8983/solr/dataimport
    -F command=full-import
```

It uses HTTP POST, which is better, as discussed much earlier.

Other boolean parameters named `clean`, `commit`, and `optimize` may accompany the parameter that defaults to `true`, if not present. Clean is specific to DIH, and it means that before running the import, it will remove all of the existing data.

Two other useful commands are `reload-config` and `abort`. The first will reload the DIH configuration file, which is useful for picking up small changes. The second will cancel any existing imports in progress.
Indexing documents with Solr Cell

While most of this book assumes that the content you want to index in Solr is in a neatly structured data format of some kind, such as in a database table, a selection of XML files, or CSV, the reality is that we store information in the much messier world of binary formats such as PDF, Microsoft Office, or even images and music files. Your author Eric Pugh, became involved with the Solr community when a client required a search system that could ingest the thousands of PDF and Microsoft Word documents that they had produced over the years. The outgrowth of that effort, Solr Cell, distributed as a contrib module to Solr, provides a very powerful framework for indexing various binary formats.

Solr Cell is technically called the Extracting Request Handler, however the name came about because:

Grant [Ingersoll] was writing the javadocs for the code and needed an entry for the  tag and wrote out "Solr Content Extraction Library", as the contrib directory is named "extraction". This then lead to an "acronym": Solr CEL, which then gets mashed to: Solr Cell! Hence, the project name is "Solr Cell"! It's also appropriate because a Solar Cell's job is to convert the raw energy of the Sun to electricity, and this contrib's module is responsible for converting the "raw" content of a document to something usable by Solr.

We'll look at how to leverage Solr Cell for extracting karaoke song lyrics from MIDI files. Just think, you can build a Solr powered index of all your favorite karaoke songs! The complete reference material for Solr Cell is available at http://wiki.apache.org/solr/ExtractingRequestHandler.

Extracting binary content

Every binary format is different, and all of them provide different types of metadata, as well as different methods of extracting content. The heavy lifting of providing a single API to an ever expanding list of binary/structured formats is left up to Apache Tika:

Apache Tika is a toolkit for detecting and extracting metadata and structured text content from various documents using existing parser libraries.
Indexing Data

Tika supports a wide variety of formats, from the predictable to the unexpected. Some of the key formats supported are Adobe PDF, Microsoft Office including Word, Excel, PowerPoint, and Visio, and Outlook. Other formats that are supported include extracting metadata from images such as JPG, GIF, and PNG, as well as from various audio formats such as MP3, MIDI, and Wave audio. Tika itself does not attempt to parse the individual binary formats. Instead, it delegates the parsing to various third party libraries, while providing a high level stream of SAX events as the documents are parsed.

Solr Cell is a fairly thin wrapper consisting of a SolrContentHandler that consumes the SAX events and builds the SolrInputDocument from the fields that are specified to be extracted from the binary document.

There are some not so obvious things to keep in mind when indexing binary documents:

- You can supply any kind of supported document to Tika, and the AutoDetectParser will attempt to discover the correct MIME type of the document. Alternatively, you can supply a stream.type parameter to specify which parser to use.
- The default SolrContentHandler that is used by Solr Cell is fairly simplistic. You may find that you need to perform extra massaging of the data being indexed beyond what Solr Cell offers to reduce the junk data being indexed. Subclass createFactory() method of ExtractingRequestHandler to return your own custom SolrContentHandler.
- Remember that as you are indexing you are potentially sending large binary files over the wire that must then be parsed in server memory, which can be very slow. If you are looking to only index metadata, then it may make sense to write your own parser using Tika directly, extract the metadata, and post that across to the server. See the Indexing HTML in Solr section in Chapter 8 for an example of parsing out metadata from an archive of a web site and posting the data through SolrJ.
- You need to supply to Solr the various dependent JAR files that Tika requires to parse the documents. Put them with the Solr Cell JAR (named something like apache-solr-cell-1.4.jar) in <solr-home>/lib.

Tika has only recently become a full fledged project and has already had a couple of releases. You can learn more about Tika from the web site at http://lucene.apache.org/tika/.
Configuring Solr

In `/examples/cores/karaoke/conf/solrconfig.xml` lies the request handler for parsing binary documents:

```xml
<requestHandler name="/update/extract"
class="org.apache.solr.handler.extraction.ExtractingRequestHandler">
  <lst name="defaults">
    <str name="map.Last-Modified">last_modified</str>
    <str name="uprefix">metadata_</str>
  </lst>
</requestHandler>
```

Here we can see that the Tika metadata attribute `Last-Modified` is being mapped to the Solr field `last_modified`, assuming we are provided that Tika attribute. The parameter `uprefix` is specifying the prefix to use when storing any Tika fields that don't have a corresponding matching Solr field.

In order to use Solr Cell, we placed the Solr Cell JAR in the `.examples/cores/karaoke/lib/` directory, because it is not included in `solr.war`. The JAR files placed in this `lib` directory are available only to the karaoke core. To share across all cores add to `.examples/cores/lib/` and by specifying it as the shared lib in `.examples/cores/solr.xml`:

```xml
<solr persistent="false" sharedLib="lib"/>
```

For this example, we are parsing `.kar` karaoke files that are recorded in the MIDI format using the standard Java package `javax.audio.midi`. However, we have also put other JAR dependencies of Solr Cell such as `pdfbox`, `poi`, and `icu4j` in `.lib`.

Extracting karaoke lyrics

We are now ready to extract karaoke lyrics by posting MIDI files to our Solr `/karaoke/update/extract` handler. Some classic ABBA tunes for your enjoyment are available in the `.examples/appendix/karaoke/songs/` directory, gratefully sourced from FreeKaraoke at `http://www.freekaraoke.com/`. In order to index the song `Angel Eyes` from the command line using `curl`, the simplest command to run is:

```bash
```
Indexing Data

Don't forget to commit your changes:

```bash
```

You can also trigger a commit when indexing content by appending `commit=true` to the URL, however this is an inefficient approach if you are indexing many documents.

We have a single `map.content=text` parameter that specifies the default field for content extracted from the source. In this case, `angeleys.kar` should be stored in the Solr field `text`. Now go look for the results at `http://localhost:8983/solr/karaoke/select/?q=*:*`. You should see:

```xml
<result name="response" numFound="1" start="0">
  <doc>
    <arr name="text">
      <str>
        Angel Eyes by Abba sequenced by Michael Boyce
tinker@worldnet.att.net Soft karaoke@KMIDI KARAOKE
        FILEWords@LENGL@TAngel Eyes@TABBA\Last night I was taking a walk along the river/And I saw him together with a young girl/And the look that he gave me shiver/'Cause he always used ...\n      </str>
    </arr>
  </doc>
</result>
```

You've now indexed information about the song and the lyrics in the text field that forms the textual content of the MIDI file. However, what about the metadata, for the MIDI file that Tika also exposes? Well, this is where dynamic fields come in very handy. Every binary format has a set of metadata that to a varying extent overlaps with other formats. Fortunately it is very easy to specify to Solr Cell how you would want to map metadata by using the `uprefix` property. We specify that all of the `metadata_*` fields should be created using dynamic fields in `schema.xml`:

```xml
<dynamicField name="metadata_*" type="string" indexed="true" stored="true" multiValued="false"/>
```

Since handling metadata properly is something we want to standardize on, we add to the configuration element in `solrconfig.xml`:

```xml
<str name="map.Last-Modified">last_modified</str>
<str name="uprefix">metadata_</str>
```

Notice that the & values in the URL are escaped with backslashes: `\&`. Forgetting to escape special characters is a common issue when working with curl.
When you search for all documents, you should see indexed metadata for *Angel Eyes*, prefixed with `metadata:`

```xml
<str name="metadata_Content-Type">audio/midi</str>
<str name="metadata_divisionType">PPQ</str>
<str name="metadata_patches">0</str>
<str name="metadata_stream_content_type">application/octet-stream</str>
<str name="metadata_stream_name">angeleyes.kar</str>
<str name="metadata_stream_size">55677</str>
<str name="metadata_stream_source_info">file</str>
<str name="metadata_tracks">16</str>
```

Obviously, in most use cases, every time you index the same file you don't want to get a new document. If your schema has a `uniqueKey` field defined such as `id`, then you can provide a specific ID by passing a literal value using `literal.id=34`. Each time you index the file using the same ID, it will delete and insert that document. However, that implies that you have the ability to manage IDs through some third party system like a database. If you want to use the metadata, such as the `stream_name` provided by Tika to provide the key, then you just need to map that field using `map.stream_name=id`. To make the example work, update `/examples/cores/karaoke/schema.xml` to specify `<uniqueKey>id</uniqueKey>`.

```
>> curl 'http://localhost:8983/solr/karaoke/update/extract?map.content=text&map.stream_name=id'  -F "file=@angeleyes.kar"
```

This of course assumes that you've defined `<uniqueKey>id</uniqueKey>` to be of type string, not a number.

### Indexing richer documents

Indexing karaoke lyrics from MIDI files is also a fairly trivial example. We basically just strip out all of the contents, and store them in the Solr text field. However, indexing other types of documents, such as PDFs, can be a bit more complicated. Let's look at *Take a Chance on Me*, a complex PDF file that explains what a Monte Carlo simulation is, while making lots of puns about the lyrics and titles of songs from ABBA. View `/examples/appendix/karaoke/mccm.pdf`, and you will see a complex PDF document with multiple fonts, background images, complex mathematical equations, Greek symbols, and charts. However, indexing that content is as simple as the prior example:

```
>> curl 'http://localhost:8983/solr/karaoke/update/extract?map.content=text&map.stream_name=id&commit=true'  -F "file=@mccm.pdf"
```
Indexing Data

If you do a search for the document using the filename as the id via http://localhost:8983/solr/karaoke/select/?q=id:mccm.pdf, then you'll also see that the last_modified field that we mapped in solrconfig.xml is being populated. Tika provides a Last-Modified field for PDFs, but not for MIDI files:

```xml
<doc>
  <arr name="id">
    <str>mccm.pdf</str>
  </arr>
  <arr name="last_modified">
    <str>Sun Mar 03 15:55:09 EST 2002</str>
  </arr>
  <arr name="text">
    <str>
      Take A Chance On Me
    </str>
  </arr>
</doc>
```

So with these richer documents, how can we get a handle on the metadata and content that is available? Passing extractOnly=true on the URL will output what Solr Cell has extracted, including metadata fields, without actually indexing them:

```xml
<response>
  ...
  <lst name="mccm.pdf_metadata">
    <arr name="stream_source_info"><str>file</str></arr>
    <arr name="subject"><str>Monte Carlo Condensed Matter</str></arr>
    <arr name="Last-Modified"><str>Sun Mar 03 15:55:09 EST 2002</str></arr>
    <arr name="creator"><str>PostScript PDriver module 4.49</str></arr>
    <arr name="title"><str>Take A Chance On Me</str></arr>
    <arr name="stream_content_type"><str>application/octet-stream</str></arr>
    <arr name="created"><str>Sun Mar 03 15:53:14 EST 2002</str></arr>
    <arr name="stream_size"><str>378454</str></arr>
    <arr name="stream_name"><str>mccm.pdf</str></arr>
  </lst>
</response>
```
At the top in an XML node called 
\texttt{<str name="mccm.pdf"/>} is the content extracted
from the PDF as an XHTML document. As it is XHTML wrapped in another separate
XML document, the various \texttt{<and>} tags have been escaped: \texttt{&lt;div&gt;}. If you cut
and paste the contents of \texttt{<str/>} node into a text editor and convert the
\texttt{&lt; and
\&gt; to >}, then you can see the structure of the XHTML document that is indexed.

Below the contents of the PDF, you can also see a wide variety of PDF
document-specific metadata fields, including subject, title, and creator, as
well as metadata fields added by Solr Cell for all imported formats, including
\texttt{stream_source_info}, \texttt{stream_content_type}, \texttt{stream_size}, and the
already-seen \texttt{stream_name}.

So why would we want to see the XHTML structure of the content? The answer
is in order to narrow down our results. We can use \texttt{xPath} queries through the
\texttt{ext.xpath} parameter to select a subset of the data to be indexed. To make up an
arbitrary example, let's say that after looking at \texttt{mccm.html} we know we only want
the second paragraph of content to be indexed:

\begin{verbatim}
content=text&map.div=divs_s&capture=div&captureAttr=true&xpath=\//\//xhtml:
p[1]' -F "file=@mccm.pdf"
\end{verbatim}

We now have only the second paragraph, which is the summary of what the
document \texttt{Take a Chance on Me} is about.

---

**Binary file size**

\textit{Take a Chance on Me} is a 372 KB file stored at \texttt{./examples/appendix/}
karaoke/mccm.pdf, and it highlights one of the challenges of using
Solr Cell. If you are indexing a thousand PDF documents that each
average 372 KB, then you are shipping 372 megabytes over the wire,
assuming the data is not already on Solr's file system. However, if you
extract the contents of the PDF on the client side and only send that over
the web, then what is sent to the Solr text field is just 5.1 KB. Look at
\texttt{./examples/appendix/karaoke/mccm.txt} to see the actual text
extracted from \texttt{mccm.pdf}. Generously assuming that the metadata adds
an extra 1 KB of information, then you have a total content sent over the
wire of 6.1 megabytes ((5.1 KB + 1.0 KB) \times 1000).

Solr Cell offers a quick way to start indexing that vast amount of
information stored in previously inaccessible binary formats without
resorting to custom code per binary format. However, depending on the
files, you may be needlessly transmitting a lot of data, only to extract a
small portion of text. Moreover, you may find that the logic provided by
Solr Cell for parsing and selecting just the data you want may not be
rich enough. For these cases you may be better off building a dedicated
client-side tool that does all of the parsing and munging you require.
Summary

At this point, you should have a schema that you believe will suit your needs, and you should know how to get your data into it. From Solr's native XML to CSV to databases to rich documents, Solr offers a variety of possibilities to ingest data into the index. Chapter 8 will discuss some additional choices for importing data. In the end, usually one or two mechanisms will be used. In addition, you can usually expect the need to write some code, perhaps just a simple bash or ant script to implement the automation of getting data from your source system into Solr.

Now that we've got data in Solr, we can finally get to querying it. The next chapter will describe Solr/Lucene's query syntax in detail, which includes phrase queries, range queries, wildcards, boosting, as well as the description of Solr's DateMath syntax. Finally, you'll learn the basics of scoring and how to debug them. The chapters after that will get to more interesting querying topics that of course depend on having data to search with.
**Where to buy this book**


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](http://www.packtpub.com/solr-1-4-enterprise-search-server/book).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.