**Solr**

Apache Solr is an open source Enterprise Java full-text search server. It was initially conceived to be a web application for providing a wide range of full-text search capabilities, using and extending the power of the well-knownApache Lucene library. The two projects have been merged into a single development effort since 2010, with improved modularity.

Solr is designed to be a standalone web server, exposing full-text and other functionalities via its own REST-like services, which can be consumed in many different ways from nearly any platform or language.

It can be also used as an embedded framework if needed, adding some of its functionalities into our Java application by a direct call to its internal API. This is a special case: useful if you need it, for example, for using its features inside a desktop application. We will only give some suggestions on how to start programming using an embedded Solr instance, at the end of the book.

Moreover, Solr is not a database; it is very different from the relational ones, as it is designed to manage indexes of the actual data (let's say, metadata useful for searching over the actual data) and not the data itself or the relations between them. However, this distinction can be very blurry in some contexts, and Solr itself is becoming a good NoSQL solution for some specific use cases. You can also see Solr as an open and evolving platform, with integrations to external third-party libraries: for data acquisitions, language processing, document clustering, and more. We will have the chance to cite some of those advanced topics when needed though the book, to have a broader idea of the possible scenarios, looking for interesting readings.

Solr is a very powerful, flexible, mature technology, and it offers not only:

* powerful **full-text search** capabilities but also
* **autosuggestion**: This simple Google search box performs queries on a remote server while we are typing, and automatically shows us some alternative term sequence that can be used for a query and has a chance to be relevant for us.
* **advanced filtering,**
* **geocoded search,**
* **highlighting in text,**
* **faceted search**: This is a particular type of search based on classification. With Solr, we can perform faceted search automatically over our fields to gain information such as how many documents have the value London for the city field. This is useful to construct some kind of faceted navigation. This is another very familiar pattern in user experience that you probably know from having used it on e-commerce site such as Amazon.  
  In the previous screenshot you can clearly recognize some facets on the top-left corner, which is suggesting that we will find a certain number of items under a certain specific "book category". For example, we know in advance that we will find 11 items for the facet "Books: Java Programming". Then, we can decide from this information whether to narrow our search or not. In case we click on the facet, a new query will be performed, adding a filter based on the choice we implicitly made. This is exactly the way a Solr faceted search will perform a similar query. The term category here is somewhat misleading, as it seems to suggest a predefined taxonomy. But with Solr we can also obtain facets on our fields without explicitly classifying the document under a certain category.

**Understanding and using an index**

The main component in Solr is the Lucene library, a full-text search library written in Java. Since Solr hides the Lucene layer from us, we don't have to study how Lucene works in detail now; you can study it in depth later. Yet it is important to have an idea of what a Lucene index is, and how it's made. Lucene's core concepts are as follows:

* **Document**: This is the main structure used both for searches and indexes. A document is an in-memory representation of the data values we need to use for our searches. In order for this to work, every document resource consists of a collection of fields, which is the simplest data structure.
* **Field**: This has its own name and value and consists of at least one term. So every document can be seen as nothing more than a list of very simple (field name and term value) pairs. If a field is designed to be multivalued, we can save as many values as we want within the same key; otherwise, if we enter new values, the last one will simply overwrite the previous.
* **Term**: This is a basic unit for indexing. For simplicity let's imagine a single word, but the word can consist of a string of words, depending on configuration details
* A close up of a logo

  Description automatically generated**Index**: This is the in-memory structure where Lucene (and Solr) perform the searches. We can then think about a document to be a single record in the Index. From an abstract logical point of view, we can easily imagine a data structure as shown in the following figure.

The best way to understand how a generic query works is by focusing on documents and trying to imagine how to search for them. While searching for the string Solr Book in the field title, if the index has been created and the fields in our query exist, we expect Lucene to search correspondences for the name-value pair title:'Solr Book' iterating over all the existing documents currently added to the index.

These kinds of document-oriented representations are often useful, as it is a common way of representing data used by many people. However, the real internal structure adopted for storing index data (and the actual process to search over the index data) is less intuitive, and we will cover it later in this chapter

**Overview of Documents, Fields, and Schema Design**

The fundamental premise of Solr is simple. You give it a lot of information, then later you can ask it questions and find the piece of information you want. The part where you feed in all the information is called indexing or updating. When you ask a question, its called a query.

One way to understand how Solr works is to think of a loose-leaf book of recipes. Every time you add a recipe to the book, you update the index at the back. You list each ingredient and the page number of the recipe you just added. Suppose you add one hundred recipes. Using the index, you can very quickly find all the recipes that use garbanzo beans, or artichokes, or coffee, as an ingredient. Using the index is much faster than looking through each recipe one by one. Imagine a book of one thousand recipes, or one million.

Solr allows you to build an index with many different fields, or types of entries. The example above shows how to build an index with just one field, ingredients. You could have other fields in the index for the recipes cooking style, like Asian, Cajun, or vegan, and you could have an index field for preparation times. Solr can answer questions like "What Cajun-style recipes that have blood oranges as an ingredient can be prepared in fewer than 30 minutes?"

The schema is the place where you tell Solr how it should build indexes from input documents.

**How Solr Sees the World**

Sol’s basic unit of information is a document, which is a set of data that describes something. A recipe document would contain the ingredients, the instructions, the preparation time, the cooking time, the tools needed, and so on. A document about a person, for example, might contain the person’s name, biography, favorite color, and shoe size. A document about a book could contain the title, author, year of publication, number of pages, and so on

In the Solr universe, documents are composed of fields, which are more specific pieces of information. Shoe size could be a field. First name and last name could be fields.

Fields can contain different kinds of data. A name field, for example, is text (character data). A shoe size field might be a floating point number so that it could contain values like 6 and 9.5. Obviously, the definition of fields is flexible (you could define a shoe size field as a text field rather than a floating point number, for example), but if you define your fields correctly, Solr will be able to interpret them correctly and your users will get better results when they perform a query.

You can tell Solr about the kind of data a field contains by specifying its field type. The field type tells Solr how to interpret the field and how it can be queried.

When you add a document, Solr takes the information in the documents fields and adds that information to an index. When you perform a query, Solr can quickly consult the index and return the matching documents.

**Field Analysis**

Field analysis tells Solr what to do with incoming data when building an index. A more accurate name for this process would be processing or even digestion, but the official name is analysis.

Consider, for example, a biography field in a person document. Every word of the biography must be indexed so that you can quickly find people whose lives have had anything to do with ketchup, or dragonflies, or cryptography.

However, a biography will likely contains lots of words you dont care about and dont want clogging up your index—words like "the", "a", "to", and so forth. Furthermore, suppose the biography contains the word "Ketchup", capitalized at the beginning of a sentence. If a user makes a query for "ketchup", you want Solr to tell you about the person even though the biography contains the capitalized word.

The solution to both these problems is field analysis. For the biography field, you can tell Solr how to break apart the biography into words. You can tell Solr that you want to make all the words lower case, and you can tell Solr to remove accents marks.

Field analysis is an important part of a field type. Understanding Analyzers, Tokenizers, and Filters is a detailed description of field analysis.

**Solrs Schema File**

Solr stores details about the field types and fields it is expected to understand in a schema file. The name and location of this file may vary depending on how you initially configured Solr or if you modified it later.

* managed-schema is the name for the schema file Solr uses by default to support making Schema changes at runtime via the Schema API, or Schemaless Mode features. You may explicitly configure the managed schema features to use an alternative filename if you choose, but the contents of the files are still updated automatically by Solr.
* schema.xml is the traditional name for a schema file which can be edited manually by users who use the ClassicIndexSchemaFactory.
* If you are using SolrCloud you may not be able to find any file by these names on the local filesystem. You will only be able to see the schema through the Schema API (if enabled) or through the Solr Admin UI’s Cloud Screens.

Whichever name of the file in use in your installation, the structure of the file is not changed. However, the way you interact with the file will change. If you are using the managed schema, it is expected that you only interact with the file with the Schema API, and never make manual edits. If you do not use the managed schema, you will only be able to make manual edits to the file, the Schema API will not support any modifications.

Note that if you are not using the Schema API yet you do use SolrCloud, you will need to interact with schema.xml through ZooKeeper using upconfig and downconfig commands to make a local copy and upload your changes. The options for doing this are described in Solr Control Script Reference and Using ZooKeeper to Manage Configuration Files.

**Core vs Collection**

A **core** is used in a context of a standalone mode or non-distributed search using one server which we get when we have a demo setup.

A picture containing bird

Description automatically generatedA **collection** is used in Solr cloud mode or distributed search where the index is spread across several computers to handle the search traffic.

Standalone mode: only one Solr node is running.

**Index vs Inverted Index**

This section describes the process of indexing: adding content to a Solr index and, if necessary, modifying that content or deleting it. By adding content to an index, we make it searchable by Solr.

A Solr index can accept data from many different sources, including:

* XML files,
* comma-separated value (CSV) files,
* data extracted from tables in a database,
* and files in common file formats such as Microsoft Word or PDF.

Here are the three most common ways of loading data into a Solr index:

* Using the Solr Cell framework built on Apache Tika for ingesting binary files or structured files such as Office, Word, PDF, and other proprietary formats.
* Uploading XML files by sending HTTP requests to the Solr server from any environment where such requests can be generated.
* Writing a custom Java application to ingest data through Solr’s Java Client API (which is described in more detail in Client APIs). Using the Java API may be the best choice if you’re working with an application, such as a Content Management System (CMS), that offers a Java API.

Regardless of the method used to ingest data, there is a common basic data structure for data being fed into a Solr index:

* a **document** containing multiple **fields**, each with a **name** and containing **content**, which may be empty. One of the fields is usually designated as a **unique ID field** (analogous to a primary key in a database), although the use of a unique ID field is not strictly required by Solr.

If the field name is defined in the Schema that is associated with the index, then the analysis steps associated with that field will be applied to its content when the content is tokenized. Fields that are not explicitly defined in the Schema will either be ignored or mapped to a dynamic field definition (see Documents, Fields, and Schema Design), if one matching the field name exists.

**Posting example documents to the first Solr core**

First, we will cover how to write a generic, simple, and useful configuration for **Solr core** and become familiar with the number of binary files saved to the disk. We will write our new configuration from the beginning, using as few elements as we can, reducing them to essential parts in order to focus on the main concepts.

A screenshot of a cell phone

Description automatically generated**Analyzing the elements we need in Solr core**

In order to use Solr we have to create a small set of files that define what we usually call a configuration for a core. Solr core is simply a working Lucene index with Solr configuration files, which we need to learn to write.

A typical Solr core directory looks like the one shown in the following figure, which is a structure for the **basic** example I have put on the disk under the /SolrStarterBook/solr-app/chp01/ directory, after posting some documents:

We call the **Solr Home** folder on the filesystem, where a running Solr instance reads the configuration to start with. This should not be confused with the installation of Solr, as that can be on a different path.

* **Solr Home** contains one or more Solr **cores**, each containing a special **/conf** directory that holds its configurations and a **/data** directory for storing the index on the disk.

**Example** ( <https://bigwisdom.net/solr-tutorial-learn-solr-in-2-hours/> )

So far, Solr is up and running. Solr is shipped with few working examples, but for this Solr tutorial, we are going to start from scratch and create a new example. First, lets create a **document collection** called “cars”: $bin/solr create -c cars

In Solr vocabulary, we call this document collection a “**core**”. It’s an instance where we can add documents and search them. We can have as many cores as we want. You can also create a Solr core through the Admin UI under section Core Admin in the menu on your left-hand side. If everything goes well, your core is now created and stored under

…/solr-8.6.0/server/solr/cars.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

**Solr Configuration**

Solr is very easy to configure. It can be done through the Admin UI or directly through files. The most important files to know are the indexing schema file (**managed-schema / schema.xml**) and the Solr configuration file (**solrconfig.xml**). Both of them are located under server/solr/cars/conf/

* Changing manages-schema to schema.xml: When you create a core a file **managed-schema** is created in **server\solr\my\_collection\conf\**. Rename this file to **schema.xml** and restart solr, after adding   
  <schemaFactory class="ClassicIndexSchemaFactory"/> to **solrconfig.xml**.

The schema file is accessible under section Schema in the Admin UI. Under section Files, you can access to the Solr configuration file, as well as other relevant files for the configuration.

**Solr Indexing Schema**

This file is used to tell Solr how to index the documents that we want to make available for search. It’s a powerful way to make sure that Solr is indexing your content in the right way. This file contains the document fields definition and the types of these fields.

The schema contains also the field “id”, which is the unique key for each document. The “id” field is already pre-defined in every schema (<uniqueKey>id</uniqueKey>).

What’s interesting about Solr is that you can define your own types to tell Solr how the fields should be processed and indexed.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generatedA screenshot of a social media post

Description automatically generated**Schema.xml vs managed-schema**

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

A picture containing knife

Description automatically generated

**Configuring Solr Home and Solr core discovery**

The first file we need to define is **solr.xml**. This file defines the general configuration of a particular Solr Home, containing one or more cores (multicore). This turns out to be useful if we want to apply different configurations over the same example, or to play with different examples. This will help us in becoming familiar with multicore from the beginning. Steps for configuring Solr Home are as follows:

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

In this default example, all we had to do in order to enable automatic discovery of cores was add the **<solrcloud />** XML tag, omitting other configurations that can be added later when we need them.

* When Solr finds this tag, it will search all the subfolders for a **core.propertiesfile**. If a file with that name exists in a directory, Solr will try to load the corresponding core. Note that this file can even be empty, and in that case the values used are the default ones that we have used here, explicitly to fix ideas. In fact the only non-default value is the name. If the name is not defined, it will be assumed that the name of the folder containing the file is the core name

Note that if you want to hide a core from the discovery mechanism, you can simply rename the core.properties file (for example, rename it to something like core.properties.skip).

**Knowing the legacy solr.xml format**

Before Solr 4.4, the solr.xml file syntax was completely different, and this will be deprecated from Solr 5. You can still use the syntax of both the versions in the current version, but you have to expect that the legacy format will be dropped in the near future. However, many examples on the internet and books are still based on this legacy format. So it's important to be able to recognize at least the basic elements for it.

A screenshot of a cell phone

Description automatically generatedWe can reproduce the configuration using the legacy format:

* Note how in this case the presence of the core.properties file is **ignored**, and every core that we want to load must have an entry in the XML file, as shown in the preceding code example.
* As you can imagine, the legacy and current syntax are mutually exclusive. If both are used on the same configuration, an exception is thrown in order to avoid problems from the start

**Writing a simple solrconfig.xml file**

In the **solrconfig.xml** file we can define how to manage requests and data manipulation for the user. This file can contain a number of different configurations. Here we can plug-in specific components and define how they are integrated within the default workflow of the data. Their typical uses include helping API with suggestions to expose, and to customize them for different language localizations

Steps for writing a simple solrconfig.xml file are as follows:

1. Using this structure as a reference, let's write a basic sorlconfig.xml file, and save it, for example, save it under the path /SolrStarterBook/solr-app/chp02/simple/conf/.
2. A screenshot of a cell phone

   Description automatically generated
3. In a standard configuration there are a few more tags, but I have omitted them here as we don't need them at the moment.

**What just happened?**

Solr exposes various components as services, in order to provide a different kind of functionality. These components are managed with the RequestHandler components:

* **Query handler**: this is a standard handler implicitly mapped on the path /select, unless we decide to explicitly use another name. We have already seen this handler in action, as it was the one that received our first test query.
* **Update handler**: This is explicitly mapped to the path /update. This is used to receive new data to be indexed. Please note that the data to be indexed in Solr are generally called documents, and it's important to distinguish them from the original data we want to search over. A Solr document is indeed a flat, unstructured sequence of the indexed metadata we want to use to represent a specific resource in our search domain.
* **Admin handler**: This is explicitly mapped to the path /admin. This is essential for using the admin web interface and having access to statistics, debug, and so on. Somewhat related to this definition is the definition of the default query to use in the admin interface.

In a common configuration there can be other exposed services such as /ping (used for checking if a core is still correctly running), but for now I decided to omit them to keep things simple and focus on understanding how the various sections are designed.

**Writing a simple schema.xml file**

A screenshot of a cell phone

Description automatically generatedWe can define the structure of a Solr document by writing the **schema.xml** file (by defining its fields); we can also define some data manipulation strategy such as tokenizing texts in order to take care of single words instead of full phrases. The steps for writing a simple schema.xml file are as follows:

1. The focus here is on how to model the data on which we will do the searches and navigation for our specific domain of interest. We are only shaping metadata or let's say a 'projection' of data useful for our searches. A typical structure for the schema.xml file will involve the following elements
2. We can write a simple schema and save it as /SolrStarterbook/solr-app/chp02/conf/schema.xml:

A screenshot of a social media post

Description automatically generated

**What just happened?**

It is important to underlay the difference between storing the actual data and creating an indexing process over the metadata manipulated and derived (extracted, projected, filtered, and so on) from them. With Solr we usually take care of the second case, even in a special case where we can also be interested in storing the actual data, using Solr as a NoSQL database, as we will see later.

We may not necessarily be interested in describing all the data we have (for example, what we have in our databases, meaning in our whole application/domain), but only what can be relevant in the search and navigation context.

At the beginning it can look like a duplication of functionality between Solr and a relational database technology, but it is not. Solr is not designed to replace traditional relational databases. In most cases Solr is used in parallel with the relational database, to expose a simple and efficient full-text API over the DBMS data. As we will see later, the data can not only be indexed but also stored in Solr so that it's even possible to adopt it as a NoSQL store in certain cases.

You should easily recognize the essential parts of this file as follows:

* **Types**: This is used for defining a list of different data types for values. We can define strings, numeric types, or new types as we like. It's very common to have two or three different data types for handling text values shaped for different purposes, but for the moment we need to focus on the main concepts.
* **Fields**: These are an essential part of this file. Every field should declare a unique **name** and associate it with one of the **types** defined previously. It's important to understand that not every instance of a Solr document must have a value for every field; when mandatory, a field can be simply marked as **required**. This approach is very flexible; we index only the actual data values without introducing dummy empty fields when a value is not present.
* dynamicfield: By using this type we can start indexing some data without having to define the name of the field. The name will be defined by the wildcard, and it's possible to use prefixes and postfixes so that the actual name of a field is accepted at runtime, while the type should be defined. For example, when writing <dynamicField name='\*\_s' type='string' /> we can post new documents containing string values such as   
  firstName\_s='Alfredo' and surname\_s='Serafini'. This is an ideal case for prototypes, as we can work with the Solr API without defining a final schema for our data.
* **uniqueKey**: This is used to give a unique identity to a specific Solr document. It is a concept similar to a primary key for DBMS.
* **defaultSearchField**: This field is used when there is no request for a specific field. The best configuration for this is generally the field containing all the full-text tokens (for example, the destination in the copyfield definition seen earlier).
* **defaultOperator**: This is used to choose a default behavior when handling multiple tokens in the search. A query that uses and between the various words used for a search is intuitively narrowed to a small set of documents. So in most cases you will use the or operator instead, which is less restrictive and more natural for common queries. The and approach is generally useful, for example, when working with navigation filters or conducting an advanced search on large datasets.

Every field can define the following three important attributes:

* **multiValued**: If the value of this attribute is true, a Solr document can contain more than one instance of values for the field. The default value is false.
* **indexed**: If it is true, the field is used in index. Generally we will use only indexed fields, but it can be interesting to have them not indexed in certain instances, for example, if we want to save a value without using it for searches.
* **stored**: This is used to permanently save the original data value for a field, whether indexed (and used for searches) or not. Moreover during the indexing phase, a field is analyzed as defined by its type in the schema.xml file to update the index; however, it is not explicitly saved unless we decide to store it.

Imagine indexing several different synonyms of the same word using a word\_synonimmultivalued field, but storing only this specific word in a word\_original field. When the user searches for the word or one of its synonyms, all the documents produced as output will only contain the field word\_synonim, which is the only one stored

**Starting the new core**

Started our core with the following command:

* java -jar start.jar -Dsolr.solr.home=/Users/kleiner/develop/solr-8.6.0/my-solr-examples/chp02/ --module=http

Multiple exceptions were thrown due to deprecated or removed objects from Solr over the years:

1. In schema not supported since Solr 7: <**defaultSearchField**>fullText</**defaultSearchField**>
2. A picture containing knife

   Description automatically generatedChange attribute **defaultOperator** to **q.op**
3. Removed: <requestHandler name='/admin/' class='org.apache.solr.handler.admin.**AdminHandlers'** />
4. Added to solrconfix.xml: <schemaFactory class="ClassicIndexSchemaFactory"/>

A screenshot of a cell phone

Description automatically generated

With these changes Solr did start the core ‘simple’

**Defining an example document**

Save the following document with the name **docs.xml** under the …/chp02/ directory.

A screenshot of a cell phone

Description automatically generated

In this small example we can easily recognize how the different types of fields are used. We write a value for every field representing the document, and multiple lines for multivalued fields (one for each different value to be added). Note that the field with the names keyword\_string and language\_string are handled as dynamic fields in our configuration.

Finally, we directly added some text for the field fullText, even if we have defined it just as a destination for copyField and as the default field for searches. We did this to demonstrate two basic facts: the field is just a normal field, and its value will be added twice in the index according to copyField we have defined, which takes every field as a source (source='\*'). The reason for this apparently wrong behavior is that copyField concatenates values on the destination field. So if you already have text in the field you use as the destination, you should use another way to select sources. You probably would map sources by an explicit name, and in most real cases you would choose not to post any value to the destination field.

**Indexing an example document with cURL**

Now that we have defined our example document, let's index it! Steps for indexing an example document with cURL are as follows:

1. The simplest way to send it for indexing is by using the cURL command-line tool:  
   >> curl -X POST 'http://localhost:8983/solr/simple/update?commit=true&wt=json' -H 'Content-Type: text/xml' -d @docs.xml

A screenshot of a cell phone

Description automatically generated

Ajskfjaksjfkasfkajsfakjsfkajsfkajsfkajsf

aksflkasfk