Lab 3. Designing Schemas

Now that we have seen how to install Solr on our machine, Let's dive deeper and understand the nitty-gritty of Solr.

Assume that you are building a home that you have always desired. How would you start?
Will you just get all the bricks, cement, windows, doors, beams, and so on and ask the builders to start building? Nope! You would want to make sure you go through various designs based on the area that you have and decide on a design that you think will not only look good but also last long. Creating any application follows the same principle and demands proper schema design.

In this lab, we will traverse through schema design. We will understand how to design a schema using documents and fields. We will also see various field types and get an understanding of the Schema API. We will finally look at schemaless mode. How Solr works

---

The easiest way to understand how Solr works is to see how a telephone directory helps you to look something up. A telephone directory, or yellow pages as it is called in some places, is a book containing lots of phone numbers. It has lots of pages. Now, to find information in it would be a humongous task unless it had some sort of indexing and categorizing. For example, we can easily find all the restaurants by just navigating to the category of restaurants and finding the locality that we are living nearby.

Similarly, Solr can be imagined as a huge directory that has been fed data as per our requirement, and it can be queried to get the relevant data by using an appropriate search criteria that was indexed while feeding in the data. Let's have a look at the following diagram and understand how Solr search platform works:
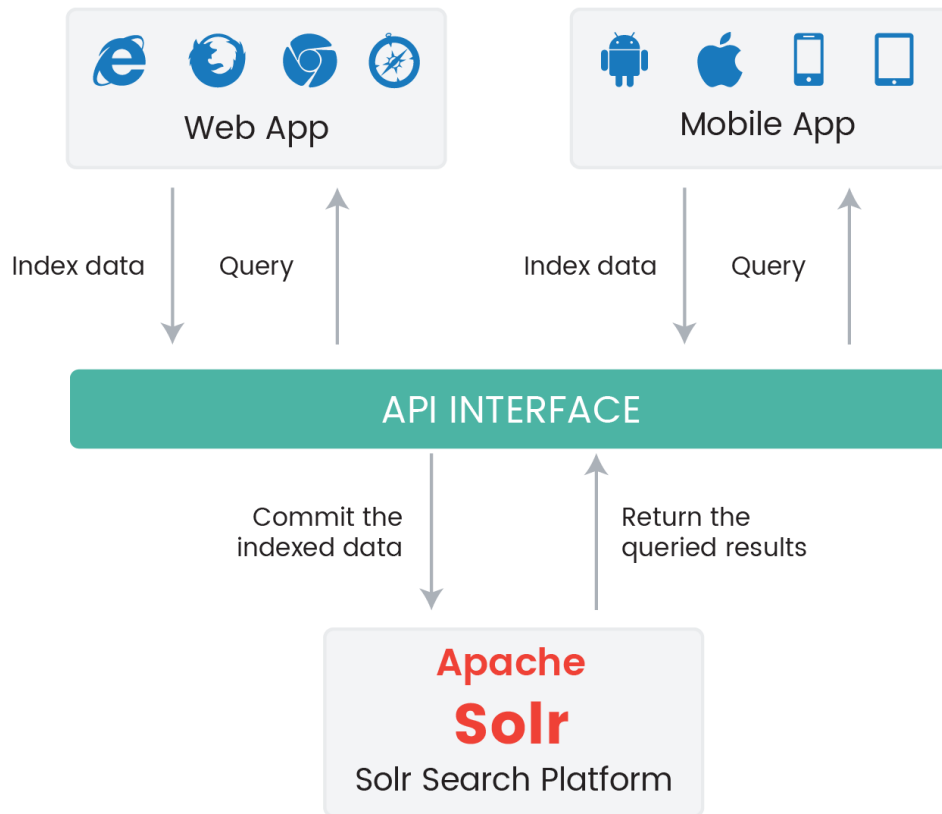
As you can see, the way to look at Solr is like this---it is basically fed with lots of information, which is correctly indexed. Then, in order to retrieve information, we query based on the index and get our results.

Suppose I am designing a data store for the world's largest online library using Solr. What I do is feed Solr with each book's information, such as the title, published date, author, price, genre, and so on. So, when I query all the books written by J.K. Rowling in the child fiction genre, it returns me my favorite [*Harry Potter*] book series.

Getting started with Solr's basics

Everything for Solr is a document, which forms the basic unit of information. Each document contains a set of fields, which can be of various types. If we take an example of a book store, each book forms a document. Now the author, publication date, and so on become fields of the document book, which can be a text or date format if we take up the example of the two fields quoted. Let's take a look at the following diagram and understand how the documents, fields are laid out in index of Solr instance:
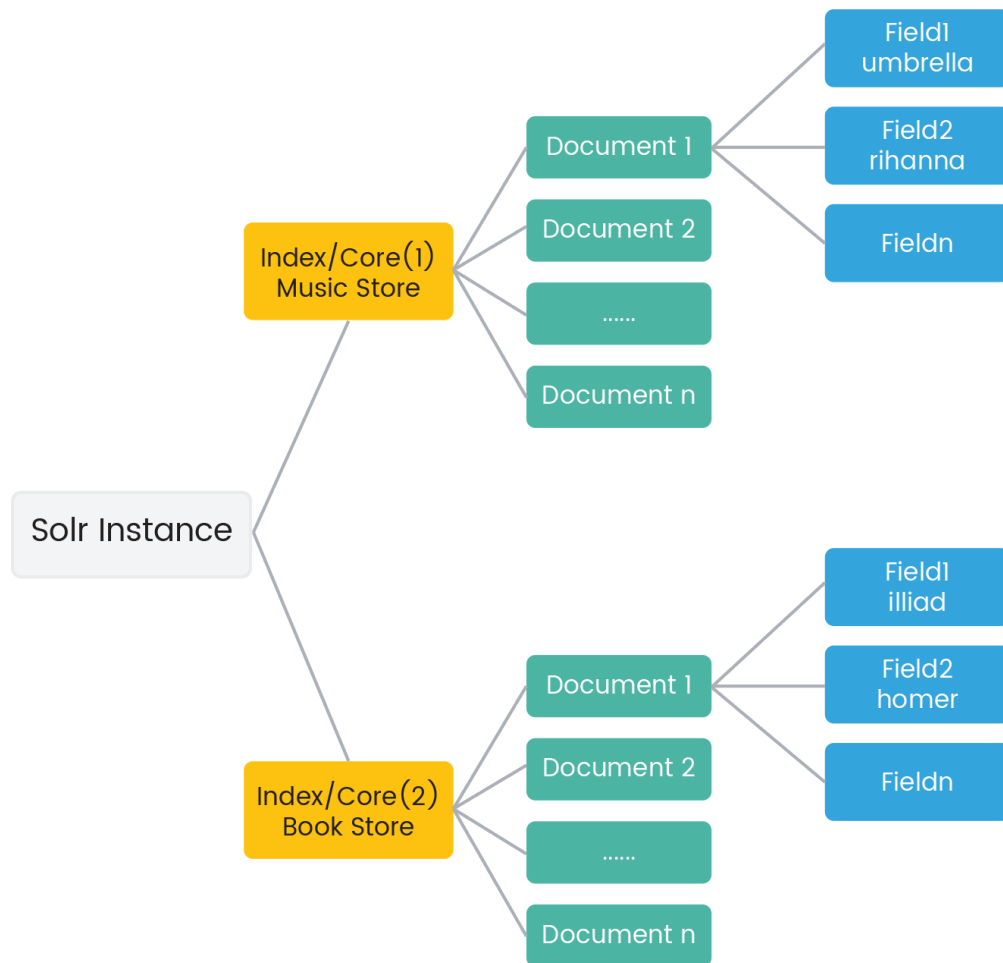
As you can see from the previous diagram, we can have as many indexes/cores in a Solr instance as we like. The first index/core in the previous case is for a music store, which can have various documents related to songs indexed in them. Each document has fields or metadata, such as singer, song title, and so on.

The schema file of Solr

All the information about fields and field types is mentioned in the schema file of Solr. Now, the schema file can be in one of the following places depending on how you have configured Solr:

- If you use `ClassicIndexSchemaFactory`, you would be manually editing the schema file which is named `schema.xml` by default
- If you decide to make schema changes at runtime using either the Schema API or schemaless mode, then the schema file is managed in the `managed-schema` file.

Note

In the case of SolrCloud, you will make changes to the schema using Solr's admin UI or Schema API if it is enabled. In such a case, you will not be able to find either `schema.xml` or `managed-schema.xml`. Understanding field types

As discussed earlier, we are able to tell Solr how it should interpret the incoming data in a field and how we can query a field using the information specified in field types.

Definitions and properties of field types

Before going to the definitions and properties, we will see what field analysis means.

What Solr should do or how it should interpret data whenever data is indexed is important. For example, a description of a book can contain lots of useless words: helping verbs such as [*is*], *[was], and [are]; pronouns such as [they], [we], and so on; and other general words such as [the], [a], [this*], and so on. Querying these words will bring all the data. Similarly what should we do with words that have capital letters?

All of these problems can be catered using field analysis to ignore common words or casing while indexing or querying. We will dive deep into field analysis in the next lab.

Now, coming back to field types, all analyses on a field are done by the field type, whether documents are indexed or a query is made on the index.

All field types are specified in `schema.xml` . A field type can have the following attributes:

- The `name` field, which is mandatory.
- The `class` field, which is also mandatory. This tells us which class to implement.
- In the case of `TextField` , you can mention description to convey what the `TextField` does.
- Based on the `Implementation` class certain field type properties which may or may not be mandatory.

The field type is defined within the `fieldType` tags. Let's take a look at a field type definition for `text_en` :

```
<fieldType name="text_en" class="solr.TextField"
  positionIncrementGap="100">
    <analyzer type="index">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <!-- in this example, we will only use synonyms at query
          time
        <filter class="solr.SynonymGraphFilterFactory"
          synonyms="index_synonyms.txt" ignoreCase="true"
          expand="false"/>
        <filter class="solr.FlattenGraphFilterFactory"/>

        -->
        <!-- Case insensitive stop word removal-->
        <filter class="solr.StopFilterFactory"
            ignoreCase="true"
            words="lang/stopwords_en.txt"
        />
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.EnglishPossessiveFilterFactory"/>
        <filter class="solr.KeywordMarkerFilterFactory"
          protected="protwords.txt"/>
        <filter class="solr.PorterStemFilterFactory"/>
    </analyzer>
    <analyzer type="query">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.SynonymGraphFilterFactory"
          synonyms="synonyms.txt" ignoreCase="true"
```

```
        expand="true"/>
        <filter class="solr.StopFilterFactory"
            ignoreCase="true"
            words="lang/stopwords_en.txt"
        />
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.EnglishPossessiveFilterFactory"/>
        <filter class="solr.KeywordMarkerFilterFactory"
          protected="protwords.txt"/>
        <filter class="solr.PorterStemFilterFactory"/>
    </analyzer>
</fieldType>
```

As you can see, the first line defines `fieldType` with name `text_en`, which implements the `solr.TextField` class. It also has an attribute, `positionIncrementGap`, which adds spaces between multi-value fields.

For example, Let's say your text contains the following tokens:

```
writer: Sandeep Nair
writer: Dharmesh Vasoya
```

Now, without any `positionIncrementGap` attribute, it is possible to bring up the results when someone searches for `Nair Dharmesh`. But with the `positionIncrementGap` attribute, we can avoid this.

We will cover the rest of the details available in `fieldType` class in detail in the next lab.

Note

You must have noticed that a class begins with `solr` in `solr.TextField`. This is a short form for the fully qualified package name `org.apache.solr.schema`.

Field type properties

All of field type is behavior generally controlled by the `fieldType` class and the optional properties:

```
<fieldType name="currency" class="solr.CurrencyFieldType"
amountLongSuffix="_l_ns" codeStrSuffix="_s_ns"
        defaultCurrency="USD" currencyConfig="currency.xml" />
```

We see that the `defaultCurrency` is `USD` and extra config-related information is specified in a file called `currency.xml`.

There are three types of properties that can be associated with a `fieldType` class:

- General properties, which can be applicable for any `fieldType` class
- Field default properties, as shown in the previous example where we default the currency value to `USD` if none is specified
- Finally, properties that are specific for a specific `fieldType` class

General properties are specified as follows:

Properties Description `name` The name of the `fieldType`. `class` The name of the class that is used to index and store the data for this type. `positionIncrementGap` Specifically for multi-value fields. It is used to specify the

distance between multiple values. This helps in preventing phrase matches that are spurious in nature. `autoGeneratePhraseQueries` This is used for `TextField`. If this value is true, Solr generates phrase queries intended for adjacent terms automatically. If it is false, then terms are supposed to be enclosed in double quotes in order for them to be treated as phrases. `enableGraphQueries` Used for only text fields. It is applicable only while querying `sow=false`. `docValuesFormat` This is used to define a custom `DocValuesFormat`. `postingsFormat` Helps in defining a custom `PostingsFormat` to be used for fields of this type.

Now Let's take a look at the field default properties. The values can be defaulted in actual fields or can be inherited from the field types.

Property	*Detail**	**Available values**	*Default value**
indexed	Tells whether to index the field or not.	true false	-
stored	The actual value of the field can be retrieved using queries if the value is true.	true false	-
sortMissingFirst and sortMissingLast	If a sort field is not present, this decides the placement of the document.	true false	-
docValues	The value of this field is placed in a column-oriented docValues structure if set to true.	true false	-
multiValued	Indicates whether a single document can have multiple values for this field type.	true false	-
omitNorms	Used to disable field length normalization and to save some memory. For all primitive field types, the value is true by default. Norms are needed only for full-text fields.	true false	-
omitTermFreqAndPositions	Omits term frequency, position, and payloads from postings of this field when true. Defaults to true for non-text fields.	true false	-
omitPositions	This is similar to omitTermFreqAndPositions; however, in this case, it preserves information about the term frequency.	true false	-
termVectors, termPositions, termOffsets, termPayloads	Solr maintains full term vectors for every single document if these properties are true. These vectors, may include position, offset, and payload information optionally for each term occurrence.	true false	-
required	Tells Solr not to accept any attempts to add a document that does not have a value for the field.	true false	-
useDocValuesAsStored	This is dependent on the docValues enabled. This is enabled if we set it to true and will allow the	true false	-

field to be ||||| returned as if ||||| it were a ||||| stored field ||||| while matching ||||| `*` ||||| in an ||||| `fl` | |||| parameter. ||| +---------------+---------------+---------------+---------------+ | `la` | This will | | `rge` | work only wh | .itemizedlist} | lse | | | | en stored="tr | - `t` | | | | ue" | rue | | | | and mul | - `fa` | | | | tiValued="fals | `lse` |||| e"`. ||||| In order ||||| not to get ||||| cached in ||||| memory, this ||||| is meant for ||||| fields that ||||| might have ||||| very large ||||| values. ||| +---------------+---------- ------+---------------+---------------+

In order to score a document in searching, Solr uses similarity. For every collection, there is one global similarity and Solr implicitly uses `BM25Similarity` . You can declare a top-level `<similarity/>` element and override it.

Note

`Best Matching (BM)` BM25 is a ranking algorithm used by search engines such as Lucene to rank matching documents according to their relevance by a given search query.

Field types available in Solr

Let's see the various field types available with Solr:

- `BinaryField` : This is intended for binary data.
- `BoolField` : This is for Boolean data. It can either be true or false. If values `1` , `t` , or `T` are encountered in the first character, then it is interpreted as true. All other values are interpreted as false.
- `CollationField` : This is used for collated sort keys, which can be used for locale-sensitive sort and range queries.
- `CurrencyFieldType` : This is used for currencies and exchange rates.

  ### Note

There is also a `CurrencyField` that does the same thing, but it is deprecated.

- `DateRangeField` : This is used for working with date ranges, as the name suggests. We will cover this in detail in the next section.
- `ExternalFileField` : This is used when values have to be pulled from an external file.
- `EnumFieldType` : This is used for enumerated sets of values.

  ### Note

There is also an `EnumField` that does the same thing, but it is now deprecated.

- `ICUCollationField` : This is similar to `CollationField` and is recommended instead of `CollationField` .
- `LatLonPointSpatialField` : This is used for multi-value for multiple points of latitude and longitude coordinate pairs. It is usually specified as latitude, longitude in that order, with a comma to separate the two.
- `PointType` : This is used when we have a single-valued n-dimensional point, and if we have to sort spatial data that is non-latitude, non-longitude or handle rare use cases.
- `PreAnalyzedField` : This is used when we have to send serialized token streams to Solr to store and index without any additional text processing.
- `RandomSortField` : This does not contain values and is used to return results in a random order.
- `SpatialRecursivePrefixTreeFieldType` : This accepts latitude, longitude strings in **well-known text** (**WKT**) format.
- `StrField` : This is intended for small fields and is not tokenized or analyzed.

- `TextField` : This is used for multiple words or tokens.
- `DatePointField` / `DoublePointField` / `FloatPointField` / `IntPointField` / `LongPointField` : These are all similar to the analogous Trie-based fields. The only difference is that they use dimensional-points-based data structures and do not require any configuration of precision steps.

Note

As of Solr 7, all the `Trie[Datatype]Fields` are deprecated: `TrieField` , `TrieDateField` , `TrieDoubleField` , `TrieFloatField` , `TrieIntField` , and `TrieLongField` .

`UUIDField` will generate a new UUID when we pass a value of `NEW` . It is recommended to use `UUIDUpdateProcessorFactory` instead of `UUIDField` to generate UUID values when using `SolrCloud` , since doing so will make each document of each replica have a unique UUID value.

Understanding date fields

As seen before, Solr's date fields such as `DatePointField` , `DateRangeField` , and `TrieDateField` (deprecated) represent dates as points in time with millisecond precision. Solr uses `DateTimeFormatter.ISO_INSTANT` for formatting and parsing:

```
YYYY-MM-DDThh:mm:ssZ
```

Let's break up the preceding date pattern. Please take a look at the break up listed as follow:

- `YYYY` is the year. An example is 1985
- `MM` is the month. For example, `02` represents February
- `DD` is the day of the month
- `T` is a literal used to separate date and time
- `hh` is the hour of the day
- `mm` is minutes
- `ss` is seconds
- `Z` is a literal used to indicate the string representation of the date in **Coordiated Universal Time** (**UTC**)

Note

No time zone can be specified and all the string representations of dates are specified in UTC.

An example would be:

```
1985-02-21T06:33:19Z
```

We can optionally add fractional seconds, but as mentioned earlier, any precision beyond milliseconds will not be considered.

Note

If we need a date prior to the year `0000` , then the date should have a leading `-` ; similarly for years after `9999` , there should be a leading `+` .

In order to express date ranges, Solr's `DateRangeField` is used. Some of the examples are shown as follows:

- `1985-02` : This represents the entire month of February 1985
- `1985-02T06` : This also adds an hour element from 6 AM to 7 AM during February 1985
- `-0002` : Since there is a leading `-` , this represents 3 BC

- `[1985-02-21 TO 1989-08-27]` : The date range between these two dates
- `[1985 TO 1985-02-21]` : From the start of 1985 until February 21, 1985
- `[* TO 1985-02-21]` : From the earliest representable time to the end of the 21st day of February 1985

Date math expressions are one more interesting format. They help by adding some quantity of time in a specified unit or rounding off the current time by a specified unit. These expressions can also be chained and they are always evaluated from left to right, like every Math expression.

Some valid expressions are as follows:

- `NOW+4DAYS` : Specifies 4 days from today.
- `NOW-6MONTHS` : Specifies 6 months before now.
- `NOW/HOUR` : Here, the slash indicates rounding. This tells us to round off to the beginning of the current hour.
- `NOW+4MONTHS+6DAYS/DAY` : This expression specifies a point in the future four months and six days from now and rounds off to the beginning of that day.

Date math can be applied between any two times and not everything has to necessarily be relative to `NOW` .

Note

The `NOW` parameter can also be used to specify an arbitrary moment in time and not necessarily the current time. It can be overridden using long-valued milliseconds since the epoch.

`DateRangeFields` also supports three relational predicates between the indexed data and the query range:

- `Intersects` (which is default)
- `Contains`
- `Within`

We can specify the predicate by querying using the `op` local parameter:

```
fq={!field f=dateRange op=Contains}[1985 TO 1989]
```

This would find documents with indexed ranges that contain the range `1985` to `1989` .

Understanding currencies and exchange rates

As the name implies, a currency field type is used for any monetary value. It supports currency conversion and exchange rates during a query.

Solr provides the following features for it:

- Range queries
- Point queries
- Sorting
- Function range queries
- Symmetric and asymmetric exchange rates
- Currency parsing

As with other field types, the `currency` field type is configured in `schema.xml` . Shown here is the default configuration:

```
<fieldType name="currency" class="solr.CurrencyFieldType"
amountLongSuffix="_l_ns" codeStrSuffix="_s_ns"
```

```
        defaultCurrency="USD" currencyConfig="currency.xml" />
```

As you can see, `name` is set as `currency` and `class` is specified as `solr.CurrencyFieldType`. We have defined the `defaultCurrency` as `INR` or Indian rupee. Also note `currencyConfig`, which says that the file location is set to `currency.xml`. This file specifies exchange rates between `INR` and other currencies.

In your open `managed-schema` file under the `SOLR_HOME/example/example-DIH/solr/solr/conf` folder, using a text editor, you will find the following dynamic field:

```
<dynamicField name="*_c" type="currency" indexed="true" stored="true"/>
```

This `dynamicField` matches any fields suffixed by `_c` and treats them as `currency` type fields.

Solr gives us the flexibility to index money fields in our native currency. We can specify `100,SGD` to index the money field in Singapore dollars:

```
<fieldType name="currency" class="solr.CurrencyFieldType"
amountLongSuffix="_l_ns" codeStrSuffix="_s_ns"
        defaultCurrency="USD" currencyConfig="currency.xml" />
```

Let's revisit the field type.

Here, you will notice one thing; there are a couple of subfield suffixes named `amountLongSuffix` and `codeStrSuffix`, which correspond to raw amount and currency code respectively. In this case, the raw amount field will make use of the `*_l_ns` dynamic field, which uses a long field type. The currency code field will make use of the `*_s_ns` dynamic field, which uses a string field type.

In the previous tag, you can also see `currencyConfig`, which refers to a file called `currency.xml`. This is used to specify exchange rates.

Solr supports two types of providers:

- `FileExchangeRateProvider`
- `OpenExchangeRatesOrgProvider`

`FileExchangeRateProvider` is the default provider. In order to use this, we specify the config file using `currencyConfig` as shown previously. The contents of `currency.xml` are as follows:

```
<?xml version="1.0" ?>
<!-- Example exchange rates file for CurrencyFieldType named "currency" in example
schema -->
<currencyConfig version="1.0">
 <rates>
 <!-- Updated from http://www.exchangerate.com/ at 2011-09-27 -->
 <rate from="USD" to="ARS" rate="4.333871" comment="ARGENTINA Peso" />
 <rate from="USD" to="AUD" rate="1.025768" comment="AUSTRALIA Dollar" />
 <rate from="USD" to="EUR" rate="0.743676" comment="European Euro" />
 <rate from="USD" to="BRL" rate="1.881093" comment="BRAZIL Real" />
 <rate from="USD" to="CAD" rate="1.030815" comment="CANADA Dollar" />
 <rate from="USD" to="CLP" rate="519.0996" comment="CHILE Peso" />
 <rate from="USD" to="CNY" rate="6.387310" comment="CHINA Yuan" />
 <rate from="USD" to="CZK" rate="18.47134" comment="CZECH REP. Koruna" />
 <rate from="USD" to="DKK" rate="5.515436" comment="DENMARK Krone" />
 <rate from="USD" to="HKD" rate="7.801922" comment="HONG KONG Dollar" />
```

```
 <rate from="USD" to="HUF" rate="215.6169" comment="HUNGARY Forint" />
 <rate from="USD" to="ISK" rate="118.1280" comment="ICELAND Krona" />
 <rate from="USD" to="INR" rate="49.49088" comment="INDIA Rupee" />
 <rate from="USD" to="XDR" rate="0.641358" comment="INTNL MON. FUND SDR" />
 <rate from="USD" to="ILS" rate="3.709739" comment="ISRAEL Sheqel" />
 <rate from="USD" to="JPY" rate="76.32419" comment="JAPAN Yen" />
 <rate from="USD" to="KRW" rate="1169.173" comment="KOREA (SOUTH) Won" />
 <rate from="USD" to="KWD" rate="0.275142" comment="KUWAIT Dinar" />
 <rate from="USD" to="MXN" rate="13.85895" comment="MEXICO Peso" />
 <rate from="USD" to="NZD" rate="1.285159" comment="NEW ZEALAND Dollar" />
 <rate from="USD" to="NOK" rate="5.859035" comment="NORWAY Krone" />
 <rate from="USD" to="PKR" rate="87.57007" comment="PAKISTAN Rupee" />
 <rate from="USD" to="PEN" rate="2.730683" comment="PERU Sol" />
 <rate from="USD" to="PHP" rate="43.62039" comment="PHILIPPINES Peso" />
 <rate from="USD" to="PLN" rate="3.310139" comment="POLAND Zloty" />
 <rate from="USD" to="RON" rate="3.100932" comment="ROMANIA Leu" />
 <rate from="USD" to="RUB" rate="32.14663" comment="RUSSIA Ruble" />
 <rate from="USD" to="SAR" rate="3.750465" comment="SAUDI ARABIA Riyal" />
 <rate from="USD" to="SGD" rate="1.299352" comment="SINGAPORE Dollar" />
 <rate from="USD" to="ZAR" rate="8.329761" comment="SOUTH AFRICA Rand" />
 <rate from="USD" to="SEK" rate="6.883442" comment="SWEDEN Krona" />
 <rate from="USD" to="CHF" rate="0.906035" comment="SWITZERLAND Franc" />
 <rate from="USD" to="TWD" rate="30.40283" comment="TAIWAN Dollar" />
 <rate from="USD" to="THB" rate="30.89487" comment="THAILAND Baht" />
 <rate from="USD" to="AED" rate="3.672955" comment="U.A.E. Dirham" />
 <rate from="USD" to="UAH" rate="7.988582" comment="UKRAINE Hryvnia" />
 <rate from="USD" to="GBP" rate="0.647910" comment="UNITED KINGDOM Pound" />
<!-- Cross-rates for some common currencies -->
 <rate from="EUR" to="GBP" rate="0.869914" />
 <rate from="EUR" to="NOK" rate="7.800095" />
 <rate from="GBP" to="NOK" rate="8.966508" />
 </rates>
</currencyConfig>
```

Updating `currency.xml` becomes tedious manual work. In order to dynamically pull the latest exchange rates from the Web, we use `OpenExchangeRatesOrgProvider`. This exchange rates from https://openexchangerates.org{.ulink}:

```
<fieldType name="currency" class="solr.CurrencyFieldType"
    amountLongSuffix="_l_ns" codeStrSuffix="_s_ns"
providerClass="solr.OpenExchangeRatesOrgProvider"
    refreshInterval="60"    ratesFileLocation=
"http://www.openexchangerates.org/api/latest.json?app_id=yourPersonalAppIdKey"/>
```

As shown in the previous tag, we have specified `providerClass` as `solr.OpenExchangeRatesOrgProvider` and specified the refresh interval as 1 hour. We have also specified the URL from which to pull the latest exchange rates.

Note

You need to register first at Open Exchange Rates to get your own personal app ID key, which you will then replace in the preceding URL.

Understanding enum fields

Just as Java has the enum data type to have something for a closed set of values, Solr has `EnumFieldType`, which lets us define closed set values. Here, the sort order is predetermined.

Defining an `EnumFieldType` is done as follows:

```
<fieldType name="genreList" class="solr.EnumFieldType" docValues="true" enumsConfig=
"enumsConfig.xml" enumName="genre"/>
```

Here, as you can see, we have defined the name as `genreList` and the class is specified as `solr.EnumFieldType`. We also have specified `enumConfig` to specify the path of the configuration file.

Last but not least, we have specified `enumName` to uniquely identify the name of the enumeration:

```
<?xml version="1.0" ?>
<enumsConfig>
    <enum name="genre">
        <value>Science Fiction</value>
        <value>Satire</value>
        <value>Drama</value>
        <value>Action</value>
        <value>Adventure</value>
        <value>Mystery</value>
        <value>Horror</value>
    </enum>
</enumsConfig>
```

The `enumsConfig` file can contain many enumeration value lists with different names as per your requirement. Take a look at the content of the enumeration file for the genres shown in the preceding code.

Field management

Once your primary work of field types setup is done, field definition is a small task. Just as with field types, the fields element of `schema.xml` holds the field definition.

Field properties

Let's first see a sample field definition:

```
<field name="weight" type="float" default="0.0" indexed="true" stored="true"/>
```

In the preceding example, we have defined a field named `weight`, whose field type is `float` with a `default` value of `0.0`. Moreover, the `indexed` as well as `stored` properties are explicitly set to `true`.

Field definitions will have these properties:

- `name`: The field name. This has to be alphanumeric and can include underscore characters. It cannot begin with a digit. Reserved names should start and end with underscores (for example, `_root_`). Every field must have a name.
- `type`: The name of the `fieldType`. All the fields should have a type.
- `default`: The default value to be used for the field.

Fields and field types share many of the optional properties here. If there are two different values of a property specified in both field and field type, then the property value specified in the field takes precedence.

Copying fields

Copying fields is used when you want to interpret a field in more than one way. Solr provides a solution to copy fields so that one can apply distinct field types for the same field.

Let's see the following element that I have copied from one of the managed-schemas available in demo Solr projects:

```
<copyField source="name" dest="text"/>
```

In the previous example, we are copying the `name` field specified in the `source` attribute to a field name `text`, which is specified in `dest`.

The actual copying of fields occurs before analysis, which makes it possible to have two fields with the same content but different analyses.

The general use case of this functionality is when we have one global search for all fields. Let's say I want to search the word `Harry` in the title and description. Then I can create a `copyField` for the title and description fields and redirect them to a common destination. Look at the following snippet:

```
<copyField source="*_e" dest="text" maxChars="10000" />
```

Here, we are specifying a wildcard pattern that matches everything that ends with `_e` and indexes them to the destination `text` field.

Note

You cannot chain `copyField`; that is, the destination of one `copyField` cannot be a part of the source of another `copyField`. The workaround is to create multiple destination fields and use the same source field.

Dynamic fields

We use dynamic fields to index fields that we do not want to explicitly define in our schema.

This feature is handy when you want to use a wild card for indexing fields, where you want to index all the fields having a certain pattern. Let's see an example of `dynamicField`:

```
<dynamicField name="*_e" type="int" indexed="true" stored="true"/>
```

As you can see, dynamic fields also have name, type, and options. However, here you can see that the name has a wildcard pattern `*_e`, which means any field with `_e` will have the same indexing. Mastering Schema API

Schema API is the one-stop shop for most operations on your schema. It provides a REST-like HTTP API for doing all these operations.

You can read, write, or delete dynamic fields, fields, copy field rules, and field types.

Note

Do not manually write any changes into the `managed-schema` file yourself. This will work only as long as you don't use Schema API. If you use Schema API by mistake, all your changes might be overwritten. So, it is highly recommend that you leave your `managed-schema` file alone.

The response of the API call is of either JSON or XML format.

Assuming that you are using the `gettingstarted` collection, the base address of API will be
`http://localhost:8983/solr/gettingstarted` .

Note

Always reindex once you use Schema API for modifications. Only then will the changes that you have applied to the schema be reflected for existing documents that are already indexed.

Schema API in detail

Let's see some of the important schema endpoints. We will do all the examples on the `gettingstarted` collection.

Schema operations

In order to see the schema, we need to use the `/schema` endpoint, `http://localhost:8983/solr/gettingstarted/schema/` .

This will retrieve the entire schema information:

```
{
  "responseHeader":{
    "status":0,
    "QTime":2},
  "schema":{
    "name":"default-config",
    "version":1.6,
    "uniqueKey":"id",
    "fieldTypes":[{
        "name":"ancestor_path",
        "class":"solr.TextField",
        "indexAnalyzer":{
          "tokenizer":{
            "class":"solr.KeywordTokenizerFactory"}},
        "queryAnalyzer":{
          "tokenizer":{
            "class":"solr.PathHierarchyTokenizerFactory",
            "delimiter":"/"}}},
      {
        "name":"binary",
        "class":"solr.BinaryField"},
      {
        "name":"boolean",
        "class":"solr.BoolField",
        "sortMissingLast":true},
      {
        "name":"booleans",
        "class":"solr.BoolField",
        "sortMissingLast":true,
        "multiValued":true},
      {
        "name":"delimited_payloads_float",
```

```
        "class":"solr.TextField",
        "indexed":true,
        "stored":false,
        "analyzer":{
          "tokenizer":{
            "class":"solr.WhitespaceTokenizerFactory"},
          "filters":[{
              "class":"solr.DelimitedPayloadTokenFilterFactory",
              "encoder":"float"}]}},
...
...
```

In order to add a field, we will use the following command:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
    "add-field":{
    "name":"song-name",
    "type":"text_general",
    "stored":true }
}' http://localhost:8983/solr/gettingstarted/schema
```

The previous code will add a `song-name` field name whose type is `text_general` to the `gettingstarted` schema.

We can also replace a field's definition. To do so we can issue the following command:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
    "replace-field":{
    "name":"song-name",
    "type":"string",
    "stored":false }
}' http://localhost:8983/solr/gettingstarted/schema
```

This will replace the definition of song with string.

Now Let's take a look at the snippet to delete the field:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
    "delete-field" : { "name":"song-name" }
}' http://localhost:8983/solr/gettingstarted/schema
```

This will delete the `song-name` field that we created just now.

Similarly, to add, delete, and replace dynamic field rules, we need to use the following endpoints:

- `add-dynamic-field`
- `delete-dynamic-field`
- `replace-dynamic-field`

We can also add, update, and delete field types using these endpoints:

- `add-field-type`
- `delete-field-type`
- `replace-field-type`

We will take a look at the syntax of making an API call to `add-field-type` as it has some additional parameters:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
    "add-field-type" : {
        "name":"song-description-field",
        "class":"solr.TextField",
        "positionIncrementGap":"100",
        "analyzer" : {
            "charFilters":[{
                "class":"solr.PatternReplaceCharFilterFactory",
                "replacement":"$1$1",
                "pattern":"([a-zA-Z])\\\\1+" }],
            "tokenizer":{
                "class":"solr.WhitespaceTokenizerFactory" },
            "filters":[{
                "class":"solr.WordDelimiterFilterFactory",
                "preserveOriginal":"0" }]}}}
}' http://localhost:8983/solr/gettingstarted/schema
```

Here, we have added a new field type for song description, which is of type `TextField` and uses `WhitespaceTokenizerFactory` as a tokenizer using a filter `WordDelimiterFilterFactory`.

Finally, in order to add or delete a copy field rule, use the following:

- `add-copy-field`
- `delete-copy-field`

Listing fields, field types, DynamicFields, and CopyField rules

In order to list all the fields, type the following URL in the browser: `http://localhost:8983/solr/gettingstarted/schema/fields`:

```
{
  "responseHeader":{
    "status":0,
    "QTime":0},
  "fields":[{
      "name":"_root_",
      "type":"string",
      "docValues":false,
      "indexed":true,
      "stored":false},
    {
      "name":"_text_",
      "type":"text_general",
      "multiValued":true,
      "indexed":true,
      "stored":false},
    {
      "name":"_version_",
      "type":"plong",
      "indexed":false,
      "stored":false},
    {
```

```
      "name":"id",
      "type":"string",
      "multiValued":false,
      "indexed":true,
      "required":true,
      "stored":true}]}
```

This will list all the fields defined. Now, to see all the field types, enter
this `http://localhost:8983/solr/gettingstarted/schema/fieldtypes` :

```
{
  "responseHeader":{
    "status":0,
    "QTime":2},
  "fieldTypes":[{
      "name":"ancestor_path",
      "class":"solr.TextField",
      "indexAnalyzer":{
        "tokenizer":{
          "class":"solr.KeywordTokenizerFactory"}},
      "queryAnalyzer":{
        "tokenizer":{
          "class":"solr.PathHierarchyTokenizerFactory",
          "delimiter":"/"}}},
    {
      "name":"binary",
      "class":"solr.BinaryField"},
    {
      "name":"boolean",
      "class":"solr.BoolField",
      "sortMissingLast":true},
    {
      "name":"booleans",
      "class":"solr.BoolField",
      "sortMissingLast":true,
      "multiValued":true},
    {
      "name":"delimited_payloads_float",
      "class":"solr.TextField",
      "indexed":true,
      "stored":false,
      "analyzer":{
        "tokenizer":{
          "class":"solr.WhitespaceTokenizerFactory"},
        "filters":[{
            "class":"solr.DelimitedPayloadTokenFilterFactory",
```

This will display all the field types. You can also see an individual field type by passing the field type name. Similarly,
to display dynamic fields and copy field rules, we can use the following endpoints
respectively, `http://localhost:8983/solr/gettingstarted/schema/dynamicfields` :

```
{
  "responseHeader":{
    "status":0,
    "QTime":1},
  "dynamicFields":[{
      "name":"*_txt_en_split_tight",
      "type":"text_en_splitting_tight",
      "indexed":true,
      "stored":true},
    {
      "name":"*_descendent_path",
      "type":"descendent_path",
      "indexed":true,
      "stored":true},
    {
      "name":"*_ancestor_path",
      "type":"ancestor_path",
      "indexed":true,
      "stored":true},
    {
      "name":"*_txt_en_split",
      "type":"text_en_splitting",
      "indexed":true,
      "stored":true},
    {
      "name":"*_txt_rev",
      "type":"text_general_rev",
      "indexed":true,
      "stored":true},
    {
      "name":"*_phon_en",
      "type":"phonetic_en",
      "indexed":true,
      "stored":true},
    {
      "name":"*_s_lower",
      "type":"lowercase",
      "indexed":true,
      "stored":true},
...
...
```

We can see a list of all copy fields using the following URL:

```
http://localhost:8983/solr/gettingstarted/schema/copyfields .
```

We will see this response:

```
{
  "responseHeader":{
    "status":0,
    "QTime":0},
  "copyFields":[]}
```

```
      "name":"*_txt_en_split_tight",
```

In order to see the schema name, use this URL:
`http://localhost:8983/solr/gettingstarted/schema/name` .

This will display the schema name's details:

```
{
  "responseHeader":{
    "status":0,
    "QTime":0},
  "name":"default-config"}
```

Similarly, you can see the schema version using the following
URL, `http://localhost:8983/solr/gettingstarted/schema/version` :

```
{
  "responseHeader":{
    "status":0,
    "QTime":0},
  "version":1.6}
```

This shows that the current schema version is `1.6` .

Likewise, in order to see the unique key, we can use this
URL, `http://localhost:8983/solr/gettingstarted/schema/uniquekey` :

```
{
  "responseHeader":{
    "status":0,
    "QTime":0},
  "uniqueKey":"id"}
```

Here we see that the `uniqueKey` of the schema is `id` .

Finally, in order to see the class name of the global similarity, we have to use the following
URL, `http://localhost:8983/solr/gettingstarted/schema/similarity` :

```
{
  "responseHeader":{
    "status":0,
    "QTime":0},
  "similarity":{
    "class":"org.apache.solr.search.similarities.SchemaSimilarityFactory"}}
```

This shows that our schema uses `SchemaSimilarityFactory` as the global similarity.

Deciphering schemaless mode

Schemaless mode is used when we want to quickly create a useful schema by indexing sample data. It does not involve any manual editing of the data.

All of its features are managed by `solrconfig.xml` .

The features that we are particularly interested in are:

- **Managed schema**: All modifications in the schema are made via Solr API at runtime using `schemaFactory`, which supports these changes.
- **Field value class guessing**: This is a technique of using a cascading set of parsers on fields that have not been seen before. It then guesses whether the field is an Integer, Long, Float, Double, Boolean, or Date.

And finally used for automatic schema field addition that is based on field value classes.

Creating a schemaless example

All of the preceding three features are already configured in the Solr bundle. To start using schemaless mode, run the following command:

```
solr start -e schemaless
```

This will start a single Solr server with the collection `gettingstarted`.

In order to see the schema fields, use the following URL, `http://localhost:8983/solr/gettingstarted/schema/fields`:

```
{
  "responseHeader":{
    "status":0,
    "QTime":66},
  "fields":[{
      "name":"_root_",
      "type":"string",
      "docValues":false,
      "indexed":true,
      "stored":false},
    {
      "name":"_text_",
      "type":"text_general",
      "multiValued":true,
      "indexed":true,
      "stored":false},
    {
      "name":"_version_",
      "type":"plong",
      "indexed":false,
      "stored":false},
    {
      "name":"id",
      "type":"string",
      "multiValued":false,
      "indexed":true,
      "required":true,
      "stored":true}]}
```

As you can see in the previous response, there are three predefined fields available, named `_text_`, `_version_`, and `_id_`.

Schemaless mode configuration

We have already discussed that the schemaless mode provides three configuration elements. In the `_default` config set, we already have these three preconfigured. Let's see how to make changes and get our own schemaless configuration.

Managed schema

The support for Managed schema is enabled by default if you don't specify anything in `solr-config.xml`.

However, if you wish to make changes, then you can explicitly add your own `schemaFactory`, as follows:

```
<schemaFactory class="ManagedIndexSchemaFactory">
    <bool name="mutable">true</bool>
    <str name="managedSchemaResourceName">managed-schema</str>
</schemaFactory>
```

In the previous code snippet, we have changed `managedSchemaResourceName` to `managed-schema`.

Field guessing

If you open `solrconfig.xml` for the new schemaless project that you have created, you will see there is a section for `UpdateRequestProcessorChain`. This is primarily used to automatically apply some operations to the documents before they get indexed and helps in field guessing. We will see some of the snippets from `solrconfig.xml` now:

```
<updateProcessor class="solr.RemoveBlankFieldUpdateProcessorFactory" name="remove-blank"/>
```

The previous plugin will remove any blanks from indexing:

```
<updateProcessor class="solr.ParseBooleanFieldUpdateProcessorFactory" name="parse-boolean"/>
 <updateProcessor class="solr.ParseLongFieldUpdateProcessorFactory" name="parse-long"/>
 <updateProcessor class="solr.ParseDoubleFieldUpdateProcessorFactory" name="parse-double"/>
 <updateProcessor class="solr.ParseDateFieldUpdateProcessorFactory" name="parse-date">
    <arr name="format">
        <str>yyyy-MM-dd'T'HH:mm:ss.SSSZ</str>
        <str>yyyy-MM-dd'T'HH:mm:ss,SSSZ</str>
        <str>yyyy-MM-dd'T'HH:mm:ss.SSS</str>
        <str>yyyy-MM-dd'T'HH:mm:ss,SSS</str>
        <str>yyyy-MM-dd'T'HH:mm:ssZ</str>
        <str>yyyy-MM-dd'T'HH:mm:ss</str>
        <str>yyyy-MM-dd'T'HH:mmZ</str>
        <str>yyyy-MM-dd'T'HH:mm</str>
        <str>yyyy-MM-dd HH:mm:ss.SSSZ</str>
        <str>yyyy-MM-dd HH:mm:ss,SSSZ</str>
        <str>yyyy-MM-dd HH:mm:ss.SSS</str>
        <str>yyyy-MM-dd HH:mm:ss,SSS</str>
        <str>yyyy-MM-dd HH:mm:ssZ</str>
        <str>yyyy-MM-dd HH:mm:ss</str>
        <str>yyyy-MM-dd HH:mmZ</str>
        <str>yyyy-MM-dd HH:mm</str>
```

```
            <str>yyyy-MM-dd</str>
        </arr>
    </updateProcessor>
```

Here, you can see that we have added many update request processors to parse various field types. In the case of dates, we can also specify various patterns of a date that we can interpret:

```
<updateProcessor class="solr.AddSchemaFieldsUpdateProcessorFactory" name="add-schema-
fields">
  <lst name="typeMapping">
        <str name="valueClass">java.lang.String</str>
        <str name="fieldType">text_general</str>
        <lst name="copyField">
            <str name="dest">*_str</str>
            <int name="maxChars">256</int>
        </lst>
        <!-- Use as default mapping instead of defaultFieldType -->
        <bool name="default">true</bool>
    </lst>
    <lst name="typeMapping">
        <str name="valueClass">java.lang.Boolean</str>
        <str name="fieldType">booleans</str>
    </lst>
    <lst name="typeMapping">
        <str name="valueClass">java.util.Date</str>
        <str name="fieldType">pdates</str>
    </lst>
    <lst name="typeMapping">
        <str name="valueClass">java.lang.Long</str>
        <str name="valueClass">java.lang.Integer</str>
        <str name="fieldType">plongs</str>
    </lst>
    <lst name="typeMapping">
        <str name="valueClass">java.lang.Number</str>
        <str name="fieldType">pdoubles</str>
    </lst>
  </updateProcessor>
```

In the preceding snippet, you can see that we have assigned a field type to the fields that we parsed. The default field type is String, as you can see highlighted in bold. We also made the copy field rule to copy the data to `text_general_str` with a max of `256` characters:

```
 <updateRequestProcessorChain name="add-unknown-fields-to-the-schema"
default="${update.autoCreateFields:true}"
 processor="uuid,remove-blank,field-name-mutating,parse-boolean,parse-long,parse-
double,parse-date,add-schema-fields">
     <processor class="solr.LogUpdateProcessorFactory"/>
     <processor class="solr.DistributedUpdateProcessorFactory"/>
     <processor class="solr.RunUpdateProcessorFactory"/>
 </updateRequestProcessorChain>
```

Finally, we add an `updateRequestProcessChain` to add all the predefined processors and make it default.
Summary

In this lab, we got an overview of how Solr works and saw schema design. We then jumped into Solr field types and saw how to define fields, copy fields, and create dynamic fields. We moved on to the Schema API, and finally we saw what schemaless mode is all about.

In the next lab, we will get our hands dirty and learn all about analyzers, tokenizers, and filters.