ElasticSearch Cookbook  Second Edition

This book will guide you through the complete ElasticSearch ecosystem. From choosing the correct transport layer and communicating with the server to creating and customizing internal actions, you will develop an in-depth knowledge of the implementation of the ElasticSearch architecture.

After creating complex queries and analytics, mapping, aggregation, and scripting, you will master the integration of ElasticSearch’s functionality in user-facing applications and take your knowledge one-step further by building custom plugins, developing tailored mapping, executing powerful analytics, and integrating with Python and Java applications.

What this book will do for you...

- Make ElasticSearch work for you by choosing the best cloud topology and powering it with plugins
- Develop tailored mapping to take full control of index steps
- Build complex queries through managing indices and documents
- Optimize search results through executing analytics aggregations
- Manage rivers (SQL, NoSQL, and web-based) to synchronize and populate cross-source data
- Develop web interfaces to execute key tasks
- Monitor the performance of the cluster and nodes

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Over 130 advanced recipes to search, analyze, deploy, manage, and monitor data effectively with ElasticSearch
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started'
- A synopsis of the book’s content
- More information on ElasticSearch Cookbook Second Edition

About the Author

Alberto Paro is an engineer, project manager, and software developer. He currently works as a CTO at Big Data Technologies and as a freelance consultant on software engineering for Big Data and NoSQL solutions. He loves to study emerging solutions and applications mainly related to Big Data processing, NoSQL, natural language processing, and neural networks.

He began programming in BASIC on a Sinclair Spectrum when he was 8 years old, and to date, has collected a lot of experience using different operating systems, applications, and programming.

In 2000, he graduated in computer science engineering at Politecnico di Milano with a thesis on designing multiuser and multidevice web applications. He assisted professors at the university for about a year. He then came in contact with The Net Planet Company and loved their innovative ideas; he started working on knowledge management solutions and advanced data mining products. In summer 2014, his company was acquired by a Big Data technologies company, where he currently works mainly using Scala and Python on state-of-the-art big data software (Spark, Akka, Cassandra, and YARN). In 2013, he started freelancing as a consultant for Big Data, machine learning, and ElasticSearch.

In his spare time, when he is not playing with his children, he likes to work on open source projects. When he was in high school, he started contributing to projects related to the GNOME environment (gtkmm). One of his preferred programming languages is Python, and he wrote one of the first NoSQL backends on Django for MongoDB (Django-MongoDB-engine). In 2010, he began using ElasticSearch to provide search capabilities to some Django e-commerce sites and developed PyES (a Pythonic client for ElasticSearch), as well as the initial part of the ElasticSearch MongoDB river. He is the author of ElasticSearch Cookbook as well as a technical reviewer ElasticSearch Server, Second Edition, and the video course, Building a Search Server with ElasticSearch, all of which are published by Packt Publishing.
Acknowledgments

It would have been difficult for me to complete this book without the support of a large number of people.

First, I would like to thank my wife, my children, and the rest of my family for their valuable support.

On a more personal note, I’d like to thank my friend, Mauro Gallo, for his patience.

I’d like to express my gratitude to everyone at Packt Publishing who’ve been involved in the development and production of this book. I’d like to thank Amey Varangaonkar for guiding this book to completion, and Florian Hopf, Philip O’Toole, and Suvda Myagmar for patiently going through the first drafts and providing valuable feedback. Their professionalism, courtesy, good judgment, and passion for this book are much appreciated.
ElasticSearch Cookbook
Second Edition

One of the main requirements of today's applications is search capability. In the market, we can find a lot of solutions that answer this need, both in commercial as well as the open source world. One of the most used libraries for searching is Apache Lucene. This library is the base of a large number of search solutions such as Apache Solr, Indextank, and ElasticSearch.

ElasticSearch is written with both cloud and distributed computing in mind. Its main author, Shay Banon, who is famous for having developed Compass (http://www.compassproject.org), released the first version of ElasticSearch in March 2010.

Thus, the main scope of ElasticSearch is to be a search engine; it also provides a lot of features that allow you to use it as a data store and an analytic engine using aggregations.

ElasticSearch contains a lot of innovative features: it is JSON/REST-based, natively distributed in a Map/Reduce approach, easy to set up, and extensible with plugins. In this book, we will go into the details of these features and many others available in ElasticSearch.

Before ElasticSearch, only Apache Solr was able to provide some of these functionalities, but it was not designed for the cloud and does not use the JSON/REST API. In the last few years, this situation has changed a bit with the release of the SolrCloud in 2012. For users who want to more thoroughly compare these two products, I suggest you read posts by Rafał Kuć, available at http://blog.sematext.com/2012/08/23/solr-vs-elasticsearchpart-1-overview/.

ElasticSearch is a product that is in a state of continuous evolution, and new functionalities are released by both the ElasticSearch company (the company founded by Shay Banon to provide commercial support for ElasticSearch) and ElasticSearch users as plugins (mainly available on GitHub).

Founded in 2012, the ElasticSearch company has raised a total of USD 104 million in funding. ElasticSearch's success can best be described by the words of Steven Schuurman, the company's cofounder and CEO:

'It's incredible to receive this kind of support from our investors over such a short period of time. This speaks to the importance of what we're doing: businesses are generating more and more data—both user- and machine-generated—and it has become a strategic imperative for them to get value out of these assets, whether they are starting a new data-focused project or trying to leverage their current Hadoop or other Big data investments.'
ElasticSearch has an impressive track record for its search product, powering customers such as Fourquare (which indexes over 50 million venues), the online music distribution platform SoundCloud, StumbleUpon, and the enterprise social network Xing, which has 14 million members. It also powers GitHub, which searches 20 terabytes of data and 1.3 billion files, and Loggly, which uses ElasticSearch as a key value store to index clusters of data for rapid analytics of logfiles.

In my opinion, ElasticSearch is probably one of the most powerful and easy-to-use search solutions on the market. Throughout this book and these recipes, the book's reviewers and I have sought to transmit our knowledge, passion, and best practices to help readers better manage ElasticSearch.

**What This Book Covers**

*Chapter 1, Getting Started*, gives you an overview of the basic concepts of ElasticSearch and the ways to communicate with it.

*Chapter 2, Downloading and Setting Up*, shows the basic steps to start using ElasticSearch, from the simple installation to running multiple nodes.

*Chapter 3, Managing Mapping*, covers the correct definition of data fields to improve both the indexing and search quality.

*Chapter 4, Basic Operations*, shows you the common operations that are required to both ingest and manage data in ElasticSearch.

*Chapter 5, Search, Queries, and Filters*, covers the core search functionalities in ElasticSearch. The search DSL is the only way to execute queries in ElasticSearch.

*Chapter 6, Aggregations*, covers another capability of ElasticSearch: the possibility to execute analytics on search results in order to improve the user experience and drill down the information.

*Chapter 7, Scripting*, shows you how to customize ElasticSearch with scripting in different programming languages.

*Chapter 8, Rivers*, extends ElasticSearch to give you the ability to pull data from different sources such as databases, NoSQL solutions, and data streams.

*Chapter 9, Cluster and Node Monitoring*, shows you how to analyze the behavior of a cluster/node to understand common pitfalls.
Chapter 10, Java Integration, describes how to integrate ElasticSearch in a Java application using both REST and native protocols.

Chapter 11, Python Integration, covers the usage of the official ElasticSearch Python client and the Pythonic PyES library.

Chapter 12, Plugin Development, describes how to create the different types of plugins: site and native plugins. Some examples show the plugin skeletons, the setup process, and their build.
In this chapter, we will cover:

- Understanding nodes and clusters
- Understanding node services
- Managing your data
- Understanding clusters, replication, and sharding
- Communicating with ElasticSearch
- Using the HTTP protocol
- Using the native protocol
- Using the Thrift protocol

**Introduction**

To efficiently use **ElasticSearch**, it is very important to understand how it works.

The goal of this chapter is to give the readers an overview of the basic concepts of ElasticSearch and to be a quick reference for them. It’s essential to understand the basics better so that you don’t fall into the common pitfall about how ElasticSearch works and how to use it.

The key concepts that we will see in this chapter are: **node**, **index**, **shard**, **mapping/type**, **document**, and **field**.

ElasticSearch can be used both as a search engine as well as a data store.

A brief description of the ElasticSearch logic helps the user to improve performance, search quality, and decide when and how to optimize the infrastructure to improve scalability and availability.
Getting Started

Some details on data replications and base node communication processes are also explained. At the end of this chapter, the protocols used to manage Elasticsearch are also discussed.

Understanding nodes and clusters

Every instance of Elasticsearch is called a node. Several nodes are grouped in a cluster. This is the base of the cloud nature of Elasticsearch.

Getting ready

To better understand the following sections, some basic knowledge about the concepts of the application node and cluster are required.

How it works...

One or more Elasticsearch nodes can be set up on a physical or a virtual server depending on the available resources such as RAM, CPU, and disk space.

A default node allows you to store data in it to process requests and responses. (In Chapter 2, Downloading and Setting Up, we’ll see details about how to set up different nodes and cluster topologies).

When a node is started, several actions take place during its startup, such as:

- The configuration is read from the environment variables and the elasticsearch.yml configuration file
- A node name is set by the configuration file or is chosen from a list of built-in random names
- Internally, the Elasticsearch engine initializes all the modules and plugins that are available in the current installation

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

After the node startup, the node searches for other cluster members and checks its index and shard status.
To join two or more nodes in a cluster, the following rules must be observed:

- The version of ElasticSearch must be the same (v0.20, v0.9, v1.4, and so on) or the join is rejected.
- The cluster name must be the same.
- The network must be configured to support broadcast discovery (it is configured to it by default) and they can communicate with each other. (See the Setting up networking recipe in Chapter 2, Downloading and Setting Up.)

A common approach in cluster management is to have a master node, which is the main reference for all cluster-level actions, and the other nodes, called secondary nodes, that replicate the master data and its actions.

To be consistent in the write operations, all the update actions are first committed in the master node and then replicated in the secondary nodes.

In a cluster with multiple nodes, if a master node dies, a master-eligible node is elected to be the new master node. This approach allows automatic failover to be set up in an ElasticSearch cluster.

There's more...

There are two important behaviors in an ElasticSearch node: the non-data node (or arbiter) and the data container behavior.

Non-data nodes are able to process REST responses and all other operations of search. During every action execution, ElasticSearch generally executes actions using a map/reduce approach: the non-data node is responsible for distributing the actions to the underlying shards (map) and collecting/aggregating the shard results (redux) to be able to send a final response. They may use a huge amount of RAM due to operations such as facets, aggregations, collecting hits and caching (such as scan/scroll queries).

Data nodes are able to store data in them. They contain the indices shards that store the indexed documents as Lucene (internal ElasticSearch engine) indices.

Using the standard configuration, a node is both an arbiter and a data container.

In big cluster architectures, having some nodes as simple arbiters with a lot of RAM, with no data, reduces the resources required by data nodes and improves performance in searches using the local memory cache of arbiters.

See also

- The Setting up different node types recipe in Chapter 2, Downloading and Setting Up.
Understanding node services

When a node is running, a lot of services are managed by its instance. These services provide additional functionalities to a node and they cover different behaviors such as networking, indexing, analyzing and so on.

Getting ready

Every Elasticsearch server that is running provides services.

How it works...

Elasticsearch natively provides a large set of functionalities that can be extended with additional plugins.

During a node startup, a lot of required services are automatically started. The most important are:

- **Cluster services**: These manage the cluster state, intra-node communication, and synchronization.
- **Indexing Service**: This manages all indexing operations, initializing all active indices and shards.
- **Mapping Service**: This manages the document types stored in the cluster (we'll discuss mapping in *Chapter 3, Managing Mapping*).
- **Network Services**: These are services such as HTTP REST services (default on port 9200), internal ES protocol (port 9300) and the Thrift server (port 9500), applicable only if the Thrift plugin is installed.
- **Plugin Service**: This enables us to enhance the basic Elasticsearch functionality in a customizable manner. (It's discussed in *Chapter 2, Downloading and Setting Up*, for installation and *Chapter 12, Plugin Development*, for detailed usage.)
- **River Service**: It is a pluggable service running within Elasticsearch cluster, pulling data (or being pushed with data) that is then indexed into the cluster. (We'll see it in *Chapter 8, Rivers*.)
- **Language Scripting Services**: They allow you to add new language scripting support to Elasticsearch.

Throughout this book, we'll see recipes that interact with Elasticsearch services. Every base functionality or extended functionality is managed in Elasticsearch as a service.
Managing your data

If you are going to use ElasticSearch as a search engine or a distributed data store, it's important to understand concepts of how ElasticSearch stores and manages your data.

Getting ready

To work with ElasticSearch data, a user must have basic concepts of data management and JSON data format, which is the lingua franca to work with ElasticSearch data and services.

How it works...

Our main data container is called index (plural indices) and it can be considered as a database in the traditional SQL world. In an index, the data is grouped into data types called mappings in ElasticSearch. A mapping describes how the records are composed (fields).

Every record that must be stored in ElasticSearch must be a JSON object.

Natively, ElasticSearch is a schema-less data store; when you enter records in it during the insert process it processes the records, splits it into fields, and updates the schema to manage the inserted data.

To manage huge volumes of records, ElasticSearch uses the common approach to split an index into multiple shards so that they can be spread on several nodes. Shard management is transparent to the users; all common record operations are managed automatically in the ElasticSearch application layer.

Every record is stored in only a shard; the sharding algorithm is based on a record ID, so many operations that require loading and changing of records/objects, can be achieved without hitting all the shards, but only the shard (and its replica) that contains your object.

The following schema compares ElasticSearch structure with SQL and MongoDB ones:

<table>
<thead>
<tr>
<th>ElasticSearch</th>
<th>SQL</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index (Indices)</td>
<td>Database</td>
<td>Database</td>
</tr>
<tr>
<td>Shard</td>
<td>Shard</td>
<td>Shard</td>
</tr>
<tr>
<td>Mapping/Type</td>
<td>Table</td>
<td>Collection</td>
</tr>
<tr>
<td>Field</td>
<td>Field</td>
<td>Field</td>
</tr>
<tr>
<td>Object (JSON Object)</td>
<td>Record (Tuples)</td>
<td>Record (BSON Object)</td>
</tr>
</tbody>
</table>
There's more...

To ensure safe operations on index/mapping/objects, ElasticSearch internally has rigid rules about how to execute operations.

In ElasticSearch, the operations are divided into:

- **Cluster/index operations**: All clusters/indices with active write are locked; first they are applied to the master node and then to the secondary one. The read operations are typically broadcasted to all the nodes.

- **Document operations**: All write actions are locked only for the single hit shard. The read operations are balanced on all the shard replicas.

When a record is saved in ElasticSearch, the destination shard is chosen based on:

- The `id` (unique identifier) of the record; if the `id` is missing, it is autogenerated by ElasticSearch
- If `routing` or `parent` (we'll see it in the parent/child mapping) parameters are defined, the correct shard is chosen by the hash of these parameters

Splitting an index in shard allows you to store your data in different nodes, because ElasticSearch tries to balance the shard distribution on all the available nodes.

Every shard can contain up to $2^{32}$ records (about 4.9 billion), so the real limit to a shard size is its storage size.

Shards contain your data and during search process all the shards are used to calculate and retrieve results. So ElasticSearch performance in big data scales horizontally with the number of shards.

All native records operations (such as index, search, update, and delete) are managed in shards.

Shard management is completely transparent to the user. Only an advanced user tends to change the default shard routing and management to cover their custom scenarios. A common custom scenario is the requirement to put customer data in the same shard to speed up his operations (search/index/analytics).

**Best practices**

It's best practice not to have a shard too big in size (over 10 GB) to avoid poor performance in indexing due to continuous merging and resizing of index segments.

It is also not good to over-allocate the number of shards to avoid poor search performance due to native distributed search (it works as map and reduce). Having a huge number of empty shards in an index will consume memory and increase the search times due to an overhead on network and results aggregation phases.
Chapter 1

See also

- Shard on Wikipedia

Understanding clusters, replication, and sharding

Related to shard management, there is the key concept of replication and cluster status.

Getting ready

You need one or more nodes running to have a cluster. To test an effective cluster, you need at least two nodes (that can be on the same machine).

How it works...

An index can have one or more replicas; the shards are called primary if they are part of the primary replica, and secondary ones if they are part of replicas.

To maintain consistency in write operations, the following workflow is executed:

- The write operation is first executed in the primary shard
- If the primary write is successfully done, it is propagated simultaneously in all the secondary shards
- If a primary shard becomes unavailable, a secondary one is elected as primary (if available) and then the flow is re-executed

During search operations, if there are some replicas, a valid set of shards is chosen randomly between primary and secondary to improve its performance. ElasticSearch has several allocation algorithms to better distribute shards on nodes. For reliability, replicas are allocated in a way that if a single node becomes unavailable, there is always at least one replica of each shard that is still available on the remaining nodes.
The following figure shows some examples of possible shards and replica configuration:

The replica has a cost in increasing the indexing time due to data node synchronization, which is the time spent to propagate the message to the slaves (mainly in an asynchronous way).

To prevent data loss and to have high availability, it's good to have at least one replica; so your system can survive a node failure without downtime and without loss of data.
Related to the concept of replication, there is the **cluster status** indicator that will show you information on the health of your cluster. It can cover three different states:

- Green: This shows that everything is okay
- Yellow: This means that some shards are missing but you can work on your cluster
- Red: This indicates a problem as some primary shards are missing

**Solving the yellow status...**

Mainly, yellow status is due to some shards that are not allocated.

If your cluster is in the recovery status (meaning that it's starting up and checking the shards before they are online), you need to wait until the shards' startup process ends.

After having finished the recovery, if your cluster is always in the yellow state, you may not have enough nodes to contain your replicas (for example, maybe the number of replicas is bigger than the number of your nodes). To prevent this, you can reduce the number of your replicas or add the required number of nodes. A good practice is to observe that the total number of nodes must not be lower than the maximum number of replicas present.

**Solving the red status**

This means you are experiencing lost data, the cause of which is that one or more shards are missing.

To fix this, you need to try to restore the node(s) that are missing. If your node restarts and the system goes back to the yellow or green status, then you are safe. Otherwise, you have obviously lost data and your cluster is not usable; the next action would be to delete the index/indices and restore them from backups or snapshots (if you have done them) or from other sources. To prevent data loss, I suggest having always a least two nodes and a replica set to 1 as good practice.

Having one or more replicas on different nodes on different machines allows you to have a live backup of your data, which stays updated always.

**See also**

*Setting up different node types* in the next chapter.
Communicating with ElasticSearch

You can communicate with several protocols using your ElasticSearch server. In this recipe, we will take a look at the main protocols.

Getting ready

You will need a working instance of the ElasticSearch cluster.

How it works...

ElasticSearch is designed to be used as a RESTful server, so the main protocol is the HTTP, usually on port number 9200 and above. Thus, it allows using different protocols such as native and thrift ones.

Many others are available as extension plugins, but they are seldom used, such as memcached, couchbase, and websocket. (If you need to find more on the transport layer, simply type in Elasticsearch transport on the GitHub website to search.)

Every protocol has advantages and disadvantages. It's important to choose the correct one depending on the kind of applications you are developing. If you are in doubt, choose the HTTP Protocol layer that is the standard protocol and is easy to use.

Choosing the right protocol depends on several factors, mainly architectural and performance related. This schema factorizes advantages and disadvantages related to them. If you are using any of the protocols to communicate with ElasticSearch official clients, switching from a protocol to another is generally a simple setting in the client initialization.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Advantages</th>
<th>Disadvantages</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>▶ Frequently used&lt;br▶ API is safe and has general compatibility for different versions of ES, although JSON is suggested</td>
<td>▶ HTTP overhead</td>
<td>▶ Text</td>
</tr>
<tr>
<td>Native</td>
<td>▶ Fast network layer&lt;br▶ Programmatic&lt;br▶ Best for massive indexing operations</td>
<td>▶ If the API changes, it can break the applications&lt;br▶ Requires the same version of the ES server&lt;br▶ Only on JVM</td>
<td>▶ Binary</td>
</tr>
<tr>
<td>Thrift</td>
<td>▶ Similar to HTTP</td>
<td>▶ Related to the Thrift plugin</td>
<td>▶ Binary</td>
</tr>
</tbody>
</table>
Chapter 1

Using the HTTP protocol

This recipe shows us the usage of the HTTP protocol with an example.

Getting ready

You need a working instance of the ElasticSearch cluster. Using default configuration, ElasticSearch enables port number 9200 on your server to communicate in HTTP.

How to do it...

The standard RESTful protocol is easy to integrate.

We will see how easy it is to fetch the ElasticSearch greeting API on a running server on port 9200 using different programming languages:

- In BASH, the request will be:
  
  curl -XGET http://127.0.0.1:9200

- In Python, the request will be:
  
  import urllib
  result = urllib.open("http://127.0.0.1:9200")

- In Java, the request will be:
  
  import java.io.BufferedReader;
  import java.io.InputStreamReader;
  import java.net.URL;
  ... truncated...
  try{
      // get URL content
      URL url = new URL("http://127.0.0.1:9200");
      URLConnection conn = url.openConnection();// open the stream and put it into BufferedReader
      BufferedReader br = new BufferedReader(new InputStreamReader(conn.getInputStream()));
      String inputLine;
      while ((inputLine = br.readLine()) != null){
          System.out.println(inputLine);
      }
      br.close();
      System.out.println("Done");
  }
  Catch (MalformedURLException e) {

}
In Scala, the request will be:

```scala
scala.io.Source.fromURL("http://127.0.0.1:9200", "utf-8").getLines.mkString("\n")
```

For every language sample, the response will be the same:

```
{
"ok" : true,
"status" : 200,
"name" : "Payge, Reeva",
"version" : {
 "number" : "1.4.0",
 "snapshot_build" : false
 },
"tagline" : "You Know, for Search"
}
```

**How it works...**

Every client creates a connection to the server index / and fetches the answer. The answer is a valid JSON object. You can invoke the ElasticSearch server from any language that you like.

The main advantages of this protocol are:

- **Portability**: This uses Web standards so that it can be integrated in different languages (Erlang, JavaScript, Python, Ruby, and so on) or called via a command-line application such as cURL.

- **Durability**: The REST APIs don't change often. They don't break for minor release changes as native protocol does.

- **Simple to use**: This has JSON-to-JSON interconnectivity.

- **Good support**: This has much more support than other protocols. Every plugin typically supports a REST endpoint on HTTP.

- **Easy cluster scaling**: You can simply put your cluster nodes behind an HTTP load balancer to balance the calls such as **HAProxy** or **NGinx**.

In this book, a lot of the examples are done by calling the HTTP API via the command-line cURL program. This approach is very fast and allows you to test functionalities very quickly.
There's more...

Every language provides drivers for best integration with Elasticsearch or RESTful web services.

The Elasticsearch community provides official drivers that support the most used programming languages.

Using the native protocol

ElasticSearch provides a native protocol, used mainly for low-level communication between nodes, but very useful for fast importing of huge data blocks. This protocol is available only for Java Virtual Machine (JVM) languages and commonly is used in Java, Groovy, and Scala.

Getting ready

You need a working instance of the Elasticsearch cluster; the standard port number for native protocol is 9300.

How to do it...

The following are the steps required to use the native protocol in a Java environment (we'll discuss this in depth in Chapter 10, Java Integration):

1. Before starting, we must be sure that Maven loads the Elasticsearch.jar file by adding the following code to the pom.xml file:

   `<dependency>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch</artifactId>
    <version>1.4.1</version>
   </dependency>

2. Depending on the Elasticsearch.jar file, creating a Java client is quite easy:

   ```java
   import org.elasticsearch.common.settings.ImmutableSettings;
   import org.elasticsearch.common.settings.Settings;
   import org.elasticsearch.client.Client;
   import org.elasticsearch.client.transport.TransportClient;
   ...
   Settings settings = ImmutableSettings.settingsBuilder().
   .put("client.transport.sniff", true).build();
   // we define a new settings
   // using sniff transport allows to autodetect other nodes
   Client client = new TransportClient(settings)
   ```
Getting Started

```java
.addTransportAddress(new InetSocketTransportAddress("127.0.0.1", 9300));
// a client is created with the settings
```

**How it works...**

To initialize a native client, a settings object is required, which contains some configuration parameters. The most important ones are:

- `cluster.name`: This is the name of the cluster
- `client.transport.sniff`: This allows you to sniff out the rest of the cluster and add them into its list of machines to use

With the settings object, it's possible to initialize a new client by giving an IP address and port a number (default 9300).

**There's more...**

The native protocol is the internal one used in ElasticSearch. It's the fastest protocol that is available to communicate with ElasticSearch.

The native protocol is an optimized binary and works only for JVM languages; to use this protocol, you need to include the `elasticsearch.jar` in your JVM project. Because it depends on ElasticSearch implementation, it must be the same version of ElasticSearch cluster.

For this reason, every time you update ElasticSearch, you need to update the `elasticsearch.jar` file on which it depends and if there are internal API changes, you need to update your code.

To use this protocol, you need to study the internals of ElasticSearch, so it's not as easy to use as HTTP and Thrift protocol.

Native protocol is useful for massive data import. But as ElasticSearch is mainly thought as a REST HTTP server to communicate with, it lacks support for everything that is not standard in the ElasticSearch core, such as the plugin's entry points. So using this protocol, you are unable to call entry points made by external plugins.

The native protocol seems the most easy to integrate in a Java/JVM project. However, due to its nature that follows the fast release cycles of ElasticSearch, it changes very often. Also, for minor release upgrades, your code is more likely to be broken. Thus, ElasticSearch developers wisely tries to fix them in the latest releases.
See also

- The native protocol is the most used in the Java world and it will be deeply discussed in Chapter 10, Java Integration and Chapter 12, Plugin Development

Using the Thrift protocol

Thrift is an interface definition language, initially developed by Facebook, used to define and create services. This protocol is now maintained by Apache Software Foundation.

Its usage is similar to HTTP, but it bypasses the limit of HTTP protocol (latency, handshake and so on) and it's faster.

Getting ready

You need a working instance of ElasticSearch cluster with the thrift plugin installed (https://github.com/elasticsearch/elasticsearch-transport-thrift/); the standard port for the Thrift protocol is 9500.

How to do it...

To use the Thrift protocol in a Java environment, perform the following steps:

1. We must be sure that Maven loads the thrift library adding to the pom.xml file; the code lines are:
   
   ```xml
   <dependency>
     <groupId>org.apache.thrift</groupId>
     <artifactId>libthrift</artifactId>
     <version>0.9.1</version>
   </dependency>
   ```

2. In Java, creating a client is quite easy using ElasticSearch generated classes:

   ```java
   import org.apache.thrift.protocol.TBinaryProtocol;
   import org.apache.thrift.protocol.TProtocol;
   import org.apache.thrift.transport.TSocket;
   import org.apache.thrift.transport.TTransport;
   import org.apache.thrift.transport.TTransportException;
   import org.elasticsearch.thrift.*;
   ```
Getting Started

```java
TTransport transport = new TSocket("127.0.0.1", 9500);
TProtocol protocol = new TBinaryProtocol(transport);
Rest.Client client = new Rest.Client(protocol);
transport.open();
```

3. To initialize a connection, first we need to open a socket transport. This is done with the `TSocket(host, port)` setting, using the ElasticSearch thrift standard port 9500.

4. Then the socket transport protocol must be encapsulated in a binary protocol, this is done with the `TBinaryProtocol(transport)` parameter.

5. Now, a client can be initialized by passing the protocol. The `Rest.Client` utility class and other utility classes are generated by `elasticsearch.thrift`. It resides in the `org.elasticsearch.thrift` namespace.

6. To have a fully working client, we must open the socket (`transport.open()`).

7. At the end of program, we should close the client connection (`transport.close()`).

**There's more...**

Some drivers, to connect to ElasticSearch, provide an easy-to-use API to interact with Thrift without the boulder that this protocol needs.

For advanced usage, I suggest the use of the Thrift protocol to bypass some problems related to HTTP limits, such as:

- The number of simultaneous connections required in HTTP; Thrift transport efficiently uses resources
- The network traffic is light weight because it is binary and is very compact

A big advantage of this protocol is that on the server side it wraps the REST entry points so that it can also be used with calls provided by external REST plugins.

**See also**

- For more complete reference on the Thrift ElasticSearch plugin, the official documentation is available at [https://github.com/elasticsearch/elasticsearch-transport-thrift/](https://github.com/elasticsearch/elasticsearch-transport-thrift/)
Where to buy this book