Elasticsearch 8.x Cookbook

Fifth Edition

Over 180 recipes to perform fast, scalable, and reliable searches for your enterprise

Alberto Paro

Elasticsearch 8.x Cookbook

Fifth Edition

Over 180 recipes to perform fast, scalable, and reliable searches for your enterprise

Alberto Paro

BIRMINGHAM—MUMBAI

Elasticsearch 8.x Cookbook

Fifth Edition

Publishing Product Manager: Devika Battike
Senior Editor: Nathanya Dias
Content Development Editor: Sean Lobo
Technical Editor: Rahul Limbachiya
Copy Editor: Safis Editing
Project Coordinator: Aparna Ravikumar Nair
Proofreader: Safis Editing
Indexer: Manju Arasan
Production Designer: Ponraj Dhandapani
Marketing Coordinator: Priyanka Mhatre

First published: December 2013
Second edition: January 2015
Third edition: February 2017
Fourth edition: April 2019
Fifth edition: May 2022

Production reference: 1280422

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-80107-981-5

www.packt.com

Contributors

About the author

Alberto Paro is an engineer, manager, and software developer. He currently works as the technology architecture delivery associate director of the Accenture Cloud First data and AI team in Italy. He loves to study emerging solutions and applications, mainly related to cloud and big data processing, NoSQL, **Natural Language Processing** (**NLP**), software development, and machine learning. In 2000, he graduated in computer science engineering from Politecnico di Milano. Then, he worked with many companies, mainly using Scala/Java and Python on knowledge management solutions and advanced data mining products, using state-of-the-art big data software. A lot of his time is spent teaching others how to effectively use big data solutions, NoSQL data stores, and related technologies.

About the reviewers

Kyle Davis is the senior developer advocate with OpenSearch and Open Distro for Elasticsearch at **Amazon Web Services** (**AWS**). Kyle has a long history of working in software development, starting in the late 1990s. His experience runs the gamut from frontend development to microcontrollers, but his most passionate area of interest is NoSQL databases. He has blogged and presented extensively about technology and is the author of *Redis Microservices for Dummies*. Kyle is based out of Edmonton, Alberta, Canada.

Mahipalsinh Rana is currently **chief technology officer** (**CTO**) of Inexture Solutions LLP. At Inexture, he specializes in enterprise searching, Python, Java, and ML/AI. He has 15 years of experience. His stint with search technologies started in 2010 when he started working with Solr. He then started working with Elastic and has done various large-scale implementations and consultations. At the start of his career, he worked for Sun Microsystems, where he worked on **internationalization** (**i18n**). He likes exploring emerging technology trends such as NLP and intuitive searching for e-commerce. He plans to develop a search engine for people who are still in the early stages of technological advancement to provide them with information at ease. He has also worked on *Liferay Beginner's Guide* by Packt.

Arpit Dubey is a big data engineer with over 14 years of experience in building large-scale, data-intensive applications. He has experience in envisioning enterprise-wide data strategies, roadmaps, and architecture for large internet companies, with varied use cases. He specializes in building event-driven architectures and real-time analytical solutions, using distributed systems such as Kafka, Flink, Spark, the Hadoop stack, NoSQL databases, and graph databases. He has been an active public speaker on various technology topics and has spoken at Kafka Summit, Druid Summit, and several other technology meetups.

I would like to thank my entire family for always being my guiding light for every path I choose and every step I take.

16
Plugin Development

Elasticsearch is designed to be extended with plugins to improve its capabilities. In the previous chapters, we installed and used many of them (new queries, REST endpoints, and scripting plugins).

Plugins are application extensions that can add many features to Elasticsearch. They can have several usages, including the following:

- Adding a new scripting language (that is, Python and JavaScript plugins)
- Adding new aggregation types
- Adding a new ingest processor
- Extending Lucene-supported analyzers and tokenizers
- Using native scripting to speed up the computation of scores, filters, and field manipulation
- Extending node capabilities, for example, creating a node plugin that can execute your logic
- Monitoring and administering clusters

In this chapter, the Java language will be used to develop a native plugin, but it is possible to use any **Java virtual machine** (**JVM**) language that generates JAR files.

The standard tools for building and testing Elasticsearch components are built on top of Gradle (`https://gradle.org/`). All our custom plugins will use Gradle to build them.

In this chapter, we will cover the following recipes:

- Creating a plugin
- Creating an analyzer plugin
- Creating a REST plugin
- Creating a cluster action
- Creating an ingest plugin

Creating a plugin

Native plugins allow several aspects of the Elasticsearch server to be extended, but they require good knowledge of Java.

In this recipe, we will see how to set up a working environment to develop native plugins.

Getting ready

You need an up and running Elasticsearch installation, as we described in the *Downloading and installing Elasticsearch* recipe in *Chapter 1*, *Getting Started*.

Gradle or an **integrated development environment** (**IDE**) that supports Java programming with Gradle (version 7.3.x used in the examples), such as Eclipse, Visual Studio Code, or IntelliJ IDEA, is required. Java JDK 17 or above needs to be installed.

The code for this recipe is available in the `ch16/simple_plugin` directory.

How to do it...

Generally, Elasticsearch plugins are developed in Java using the Gradle build tool (`https://gradle.org/`) and deployed as a ZIP file.

To create a simple JAR plugin, we will perform the following steps:

1. To correctly build and serve a plugin, some files must be defined:
   - `build.gradle` and `settings.gradle` are used to define the build configuration for Gradle.
   - `LICENSE.txt` defines the plugin license.
   - `NOTICE.txt` is a copyright notice.
2. A `build.gradle` file is used to create a plugin that contains the following code:

```
buildscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath "org.elasticsearch.gradle:build-
tools:8.0.0"
  }
}
repositories {
  mavenCentral()
}
group = 'org.elasticsearch.plugin'
version = '8.0.0-SNAPSHOT'
apply plugin: 'java'
apply plugin: 'idea'
apply plugin: 'elasticsearch.esplugin'
apply plugin: 'elasticsearch.yaml-rest-test'
esplugin {
  name 'simple-plugin'
  description 'A simple plugin for ElasticSearch'
  classname 'org.elasticsearch.plugin.simple.
SimplePlugin'
  // license of the plugin, may be different than the
above license
  licenseFile rootProject.file('LICENSE.txt')
  // copyright notices, may be different than the above
notice
```

```
    noticeFile rootProject.file('NOTICE.txt')
  }
  // In this section you declare the dependencies for your
  production and test code
  // Note, the two dependencies are not really needed as
  the buildscript dependency gets them in already
  // they are just here as an example
  dependencies {
    implementation 'org.elasticsearch:elasticsearch:8.0.0'
    yamlRestTestImplementation 'org.elasticsearch.
  test:framework:8.0.0'
  }
  // ignore javadoc linting errors for now
  tasks.withType(Javadoc) {
    options.addStringOption('Xdoclint:none', '-quiet')
  }
```

3. The `settings.gradle` file is used for the project name, as follows:

```
rootProject.name = 'simple-plugin'
```

4. The `src/main/java/org/elasticsearch/plugin/simple/SimplePlugin.java` class is an example of the basic (the minimum required) code that needs to be compiled for executing a plugin, as follows:

```
package org.elasticsearch.plugin.simple;
import org.elasticsearch.plugins.Plugin;
public class SimplePlugin extends Plugin {
}
```

How it works...

Several parts make up the development life cycle of a plugin, such as designing, coding, building, and deploying. To speed up the build and deployment steps, which are common to all plugins, we need to create a `build.gradle` file.

The preceding `build.gradle` file is a standard for developing Elasticsearch plugins. This file is composed of the following:

- A `buildscript` section that depends on the Gradle building tools for Elasticsearch, as follows:

```
buildscript {
  repositories {
//    mavenLocal() // if you want use your local maven
artifact
    mavenCentral()
  }
  dependencies {
    classpath "org.elasticsearch.gradle:build-
tools:8.0.0"
  }
}
```

> **Note**
>
> The version of the plugin (that is, `"org.elasticsearch.gradle:build-tools:8.0.0"`) will be the target version of Elasticsearch.

- The group and the version of the plugin, used for artifact deployment, as follows:

```
group = 'org.elasticsearch.plugin'
version = '8.0.0-SNAPSHOT'
```

- A list of Gradle plugins that must be activated, as follows:

```
apply plugin: 'java' // for java support
apply plugin: 'idea' // for Intellij IDEA support
apply plugin: 'elasticsearch.esplugin' // Elasticsearch
plugin support
apply plugin: 'elasticsearch.yaml-rest-test' //
Elasticsearch Yaml Rest Test
```

- The core information that's needed to populate the plugin description: that is, information that is used to generate the `plugin-descriptor.properties` file that will be available in the final distribution ZIP and used by Elasticsearch to load the plugin. The most important parameter is `classname`, which is the main entry point of the plugin:

```
esplugin {
  name 'simple-plugin'
  description 'A simple plugin for ElasticSearch'
  classname 'org.elasticsearch.plugin.simple.
SimplePlugin'
  // license of the plugin, may be different than the
above license
  licenseFile rootProject.file('LICENSE.txt')
  // copyright notices, may be different than the above
notice
  noticeFile rootProject.file('NOTICE.txt')
}
```

- To compile the code, some dependencies are required, as follows:

```
dependencies {
  implements 'org.elasticsearch:elasticsearch:8.0.0'
}
```

After configuring Gradle, we can start writing the main plugin class.

Every `plugin` class must be derived from a plugin one and it must have a public scope, otherwise, it cannot be loaded dynamically from the JAR, as follows:

```
package org.elasticsearch.plugin.simple;
import org.elasticsearch.plugins.Plugin;
public class SimplePlugin extends Plugin {}
```

After having defined all the files that are required to generate a ZIP release of our plugin, it is enough to invoke the `gradle clean check` command. This command will compile the code and create a `zip` package in the `build/distributions/` directory of your project: the final ZIP file can be deployed as a plugin on your Elasticsearch cluster (see *Chapter 1*, *Installing Plugins in Elasticsearch*, as reference).

In this recipe, we configured a working environment to build, deploy, and test plugins. In the following recipes, we will reuse this environment to develop several plugin types.

There's more...

Compiling and packaging a plugin are not enough to define a good life cycle for your plugin; a test phase for testing your plugin functionalities needs to be provided.

Testing the plugin functionalities with test cases reduces the number of bugs that can affect the plugin when it's released.

The Elasticsearch test framework is designed to simplify different test scenarios such as **Unit Test** and **Integration Test** with running node instances. To enable these functionalities, make sure to put the following line in your `dependencies`:

```
dependencies {
  yamlRestTestImplementation
'org.elasticsearch.test:framework:8.0.0'
}
```

The extension of Elasticsearch for Gradle has everything to set up unit tests (`https://en.wikipedia.org/wiki/Unit_testing`) and integration tests (`https://en.wikipedia.org/wiki/Integration_testing`).

Creating an analyzer plugin

Elasticsearch provides a large set of analyzers and tokenizers to cover general needs out of the box. Sometimes, we need to extend the capabilities of Elasticsearch by adding new analyzers.

Typically, you can create an analyzer plugin when you need to do the following:

- Add standard Lucene analyzers/tokenizers that are not provided by Elasticsearch.
- Integrate third-party analyzers.
- Add custom analyzers.

In this recipe, we will add a new custom English analyzer, similar to the one provided by Elasticsearch.

Getting ready

You need an up and running Elasticsearch installation, as we described in the *Downloading and installing Elasticsearch* recipe in *Chapter 1*, *Getting Started*.

Gradle or an **integrated development environment** (**IDE**) that supports Java programming with Gradle (version 7.3.x used in the examples), such as Eclipse, Visual Studio Code, or IntelliJ IDEA, is required. Java JDK 17 or above needs to be installed.

The code for this recipe is available in the `ch16/analysis_plugin` directory.

How to do it...

An analyzer plugin is generally composed of the following two classes:

- A `Plugin` class, which implements the `org.elasticsearch.plugins.AnalysisPlugin` class
- An `AnalyzerProviders` class, which provides an analyzer

To create an analyzer plugin, we will perform the following steps:

1. The `plugin` class is similar to the ones we've seen in previous recipes, but it includes a method that returns the analyzers, as follows:

```
package org.elasticsearch.plugin.analysis;
import org.apache.lucene.analysis.Analyzer;
import org.elasticsearch.index.analysis.AnalyzerProvider;
import org.elasticsearch.index.analysis.
CustomEnglishAnalyzerProvider;
import org.elasticsearch.indices.analysis.AnalysisModule;
import org.elasticsearch.plugins.Plugin;
import java.util.HashMap;
import java.util.Map;
public class AnalysisPlugin extends Plugin implements
org.elasticsearch.plugins.AnalysisPlugin {
    @Override
    public Map<String, AnalysisModule.
AnalysisProvider<AnalyzerProvider<? extends Analyzer>>>
getAnalyzers() {
        Map<String, AnalysisModule.
AnalysisProvider<AnalyzerProvider<? extends Analyzer>>>
analyzers = new HashMap<>();
        analyzers.put(CustomEnglishAnalyzerProvider.NAME,
CustomEnglishAnalyzerProvider::getCustomEnglishAnalyz
erProvider);
```

```
        return analyzers;
    }
}
```

2. The `AnalyzerProvider` class provides the initialization of our analyzer, and passes the parameters that are provided by the settings, as follows:

```
import org.apache.lucene.analysis.en.EnglishAnalyzer;
import org.apache.lucene.analysis.CharArraySet;
import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.env.Environment;
import org.elasticsearch.index.IndexSettings;
public class CustomEnglishAnalyzerProvider extends
AbstractIndexAnalyzerProvider<EnglishAnalyzer> {
    public static String NAME = "custom_english";
    private final EnglishAnalyzer analyzer;
    public CustomEnglishAnalyzerProvider(IndexSettings
indexSettings, Environment env, String name, Settings
settings,
                                         boolean
ignoreCase) {
        super(indexSettings, name, settings);
        analyzer = new EnglishAnalyzer(
                Analysis.parseStopWords(env, settings,
EnglishAnalyzer.getDefaultStopSet(), ignoreCase),
                Analysis.parseStemExclusion(settings,
CharArraySet.EMPTY_SET));
    }
    public static CustomEnglishAnalyzerProvider
getCustomEnglishAnalyzerProvider(IndexSettings
indexSettings,

Environment env, String name,

Settings settings) {
        return new
CustomEnglishAnalyzerProvider(indexSettings, env, name,
settings, true);
    }
    @Override
```

```
    public EnglishAnalyzer get() { return this.analyzer;
} }
```

After building the plugin and installing it on an Elasticsearch server, our analyzer is accessible as any native Elasticsearch analyzer.

How it works...

Creating an analyzer plugin is quite simple. The general workflow is as follows:

1. Wrap the analyzer initialization in a provider.
2. Register the analyzer provider in the plugin.
3. In the preceding example, we registered a `CustomEnglishAnalyzerProvider` class, which extends the `EnglishAnalyzer` class:

```
public class CustomEnglishAnalyzerProvider extends
AbstractIndexAnalyzerProvider<EnglishAnalyzer> {
```

4. We need to provide a name to `analyzer`, as follows:

```
public static String NAME = "custom_english";
```

5. We instantiate a private scope Lucene analyzer (in *Chapter 2*, *Managing Mapping*, we discussed custom Lucene analyzer usage) to be provided on request with the `get` method, as follows:

```
    @Override
    public EnglishAnalyzer get() {
        return this.analyzer; }
```

6. The `CustomEnglishAnalyzerProvider` constructor can be injected via Google Guice, with settings that can be used to provide cluster defaults via index settings or `elasticsearch.yml`, as follows:

```
public CustomEnglishAnalyzerProvider(IndexSettings
indexSettings, Environment env, String name, Settings
settings, boolean ignoreCase) {
```

7. To make it work correctly, we need to set up the parent constructor via the `super` call, as follows:

```
super(indexSettings, name, settings);
```

8. Now, we can initialize the internal analyzer, which must be returned by the `get` method, as follows:

```
analyzer = new EnglishAnalyzer(
                Analysis.parseStopWords(env, settings,
EnglishAnalyzer.getDefaultStopSet(), ignoreCase),
                Analysis.parseStemExclusion(settings,
CharArraySet.EMPTY_SET));
```

This analyzer accepts the following:

- A list of stopwords that can be loaded via the settings or set by the default ones
- A list of words that must be excluded by the stemming step

9. To easily wrap the analyzer, we need to create a `static` method that can be called to create the analyzer. We'll use it in the plugin definition, as follows:

```
    public static CustomEnglishAnalyzerProvider
getCustomEnglishAnalyzerProvider(IndexSettings
indexSettings,

Environment env, String name,

Settings settings) {
        return new
CustomEnglishAnalyzerProvider(indexSettings, env, name,
settings, true); }
```

10. Finally, we can register our analyzer in the plugin. To do so, our plugin must derive from `AnalysisPlugin` so that we can override the `getAnalyzers` method, as follows:

```
    @Override
    public Map<String, AnalysisModule.
AnalysisProvider<AnalyzerProvider<? extends Analyzer>>>
getAnalyzers() {
        Map<String, AnalysisModule.
AnalysisProvider<AnalyzerProvider<? extends Analyzer>>>
analyzers = new HashMap<>();
        analyzers.put(CustomEnglishAnalyzerProvider.NAME,
CustomEnglishAnalyzerProvider::getCustomEnglishAnalyzer
Provider);
        return analyzers; }
```

The `::` operator of Java 8 allows us to provide a function that will be used for the construction of our `AnalyzerProvider`.

There's more...

A plugin extends several Elasticsearch functionalities. To provide them with this requires extending the correct plugin interface. In Elasticsearch 7.x, the following are the main plugin interfaces:

- `ActionPlugin`: This is used for REST and cluster actions.
- `AnalysisPlugin`: This is used for extending all the analysis stuff, such as **analyzers**, **tokenizers**, **tokenFilters**, and **charFilters**.
- `ClusterPlugin`: This is used to provide new deciders.
- `DiscoveryPlugin`: This is used to provide custom node name resolvers.
- `EnginePlugin`: This is used to provide a new custom engine for indices.
- `IndexStorePlugin`: This is used to provide a custom index store.
- `IngestPlugin`: This is used to provide new ingest processors.
- `MapperPlugin`: This is used to provide new mappers and metadata mappers.
- `ReloadablePlugin`: This allows you to create plugins that reload their state.
- `RepositoryPlugin`: This allows the provision of new repositories to be used in backup/restore functionalities.
- `ScriptPlugin`: This allows the provision of new scripting languages, scripting contexts, or native scripts (Java-based ones).
- `SearchPlugin`: This allows an extension of all the search functionalities: Highlighter, aggregations, suggesters, and queries.

If your plugin needs to extend more than a single functionality, it can extend from several plugin interfaces at once.

Creating a REST plugin

In the previous recipe, we read how to build an analyzer plugin that extends the query capabilities of Elasticsearch. In this recipe, we will see how to create one of the most common Elasticsearch plugins. This kind of plugin allows the standard REST calls to be extended with custom ones to easily improve the capabilities of Elasticsearch.

In this recipe, we will see how to define a **REST entry point** and create its action; in the next one, we'll see how to execute this action distributed in shards.

Getting ready

You need an up and running Elasticsearch installation, as we described in the *Downloading and installing Elasticsearch* recipe in *Chapter 1*, *Getting Started*.

Gradle or an **integrated development environment** (**IDE**) that supports Java programming with Gradle (version 7.3.x used in the examples), such as Eclipse, Visual Studio Code, or IntelliJ IDEA, is required. Java JDK 17 or above needs to be installed.

The code for this recipe is available in the `ch16/rest_plugin` directory.

How to do it...

To create a REST entry point, we need to create the action and then register it in the plugin. We will perform the following steps:

1. We create a REST `simple` action (`RestSimpleAction.java`) as follows:

```
public class RestSimpleAction extends BaseRestHandler {
    public RestSimpleAction(Settings settings,
RestController controller) {
    }
    @Override
    public List<Route> routes() {
        return unmodifiableList(asList(
                new Route(POST, "/_simple"),
                new Route(POST, "/{index}/_simple"),
                new Route(POST, "/_simple/{field}"),
                new Route(GET, "/_simple"),
                new Route(GET, "/{index}/_simple"),
                new Route(GET, "/_simple/{field}")));
            }
    @Override
    public String getName() {return "simple_rest"; }
    @Override
    protected RestChannelConsumer
prepareRequest(RestRequest request, NodeClient client) {
        final SimpleRequest simpleRequest = new
```

```
SimpleRequest(Strings.splitStringByCommaToArray(request.
param("index")));
        simpleRequest.setField(request.param("field"));
        return channel -> client.
execute(SimpleAction.INSTANCE, simpleRequest, new
RestBuilderListener<SimpleResponse>(channel) {
            @Override
            public RestResponse
buildResponse(SimpleResponse simpleResponse,
XContentBuilder builder) {
                try {
                    builder.startObject();
                    builder.field("ok", true);
                    builder.array("terms",
simpleResponse.getSimple().toArray());
                    builder.endObject();
                } catch (Exception e) {
                    onFailure(e);
                }
                return new BytesRestResponse(OK,
builder); } }); } }
```

2. We need to register it in the plugin with the following lines:

```
public class RestPlugin extends Plugin implements
ActionPlugin {
    @Override
    public List<RestHandler> getRestHandlers(Settings
settings, RestController restController,

ClusterSettings clusterSettings, IndexScopedSettings
indexScopedSettings,

SettingsFilter settingsFilter,

IndexNameExpressionResolver indexNameExpressionResolver,

Supplier<DiscoveryNodes> nodesInCluster) {
        return Arrays.asList(new
RestSimpleAction(settings, restController));
```

```
    }
    @Override
    public List<ActionHandler<? extends ActionRequest, ?
extends ActionResponse>> getActions() {
        return Arrays.asList(new
ActionHandler<>(SimpleAction.INSTANCE,
TransportSimpleAction.class)); } }
```

3. Now, we can build the plugin via `gradle clean check` and manually install the ZIP. If we restart the Elasticsearch server, we should see the plugin loaded.
4. We can test out custom REST via `curl`, as follows:

```
curl -XPUT http://127.0.0.1:9200/mytest
curl -XPUT http://127.0.0.1:9200/mytest2
curl 'http://127.0.0.1:9200/_simple?field=mytest&pretty
```

5. The result will be something similar to the following:

```
{"ok" : true,
 "terms":["mytest_[mytest2][0]","mytest_[mytest][0]"]}
```

How it works...

Adding a REST action is very easy: We need to create a `RestXXXAction` class that handles the calls.

The REST action is derived from the `BaseRestHandler` class and needs to implement the `handleRequest` method.

The constructor is very important. So, let's start by writing the following:

```
public RestSimpleAction(Settings settings, RestController
controller) {}
```

The public constructor takes the following parameters:

- `Settings`: This can be used to load custom settings for your REST action.
- `RestController`: This is used to register the advanced REST action with the controller.

The list of actions that must be registered in an override `routers` methods is as follows:

```
@Override
public List<Route> routes() {
    return unmodifiableList(asList(
            new Route(POST, "/_simple"),
            new Route(POST, "/{index}/_simple"),
            new Route(POST, "/_simple/{field}"),
            new Route(GET, "/_simple"),
            new Route(GET, "/{index}/_simple"),
            new Route(GET, "/_simple/{field}")));}
```

To register an action, a `Route` class must be instantiated with the following parameters:

- The REST method (`GET`/`POST`/`PUT`/`DELETE`/`HEAD`/`OPTIONS`)
- The URL **entry point**

After having defined the constructor, if an action is fired, the `prepareRequest` class method is called as follows:

```
RestChannelConsumer
@Override
protected RestChannelConsumer prepareRequest(RestRequest
request, NodeClient client) {
```

This method is the core of the REST action. It processes the request and sends back the result. The following parameters are passed to the method:

- `RestRequest`: This is the REST request that hits the Elasticsearch server.
- `NodeClient`: This is the client used to communicate in the cluster.

The returned value is `RestChannelConsumer`, which is a `FunctionalInterface` interface type that accepts a `RestChannel` request—it's a simple Java **Lambda**.

A `prepareRequest` method is usually composed of these phases:

- Process the REST request and build an inner Elasticsearch request object.
- Call the client with the Elasticsearch request.
- If it is okay, process the Elasticsearch response and build the resulting JSON.
- If there are errors, send back the JSON error response.

In the following example, we created `SimpleRequest` class that processes the request:

```
final SimpleRequest simpleRequest = new SimpleRequest(Strings.
splitStringByCommaToArray(request.param("index")));
simpleRequest.setField(request.param("field"));
```

As you can see, it accepts a list of indices (we split the classic comma-separated list of indices via the `Strings.splitStringByCommaToArray` helper) and we had the `field` parameter if available.

Now that we have `SimpleRequest`, we can send it to the cluster and get back `SimpleResponse` via the Lambda closure as follows:

```
return channel -> client.execute(SimpleAction.INSTANCE,
simpleRequest, new RestBuilderListener<SimpleResponse>(channel)
{
```

`client.execute` accepts an action, a request, and a `RestBuilderListener` class that maps a future response. We can now process the response via the definition of a `buildResponse` method.

`buildResponse` receives a `Response` object that must be converted to a JSON result via `XContentBuilder` as follows:

```
@Override
public RestResponse buildResponse(SimpleResponse
simpleResponse, XContentBuilder builder) {
```

The builder is the standard JSON `XContentBuilder` class that we have already seen in *Chapter 13*, *Java Integration*.

After having processed the cluster response and built the JSON, we can send the REST response as follows:

```
builder.startObject()
builder.field("ok", true);
builder.array("terms", simpleResponse.getSimple().toArray());
builder.endObject();
```

Obviously, if something goes wrong during the JSON creation, an exception must be raised, such as the following:

```
try {
    // JSON creation
```

```
} catch (Exception e) {
    onFailure(e);
}
```

We will discuss `SimpleRequest` in the next recipe.

See also

Google Guice (`https://github.com/google/guice`) is used for dependency injection. Refer to its official documentation for more insights about the dependency injection system used by Elasticsearch.

Creating a cluster action

In the previous recipe, we saw how to create a REST **entry point**, but to execute the action at the cluster level, we will need to create a cluster action.

An Elasticsearch action is generally executed and distributed in the cluster and, in this recipe, we will see how to implement this kind of action. The cluster action will be very bare; we send a string with a value to every shard and the shards echo a result string, concatenating the string with the shard number.

Getting ready

You need an up and running Elasticsearch installation, as we described in the *Downloading and installing Elasticsearch* recipe in *Chapter 1*, *Getting Started*.

Gradle or an **integrated development environment** (**IDE**) that supports Java programming with Gradle (version 7.3.x used in the examples), such as Eclipse, Visual Studio Code, or IntelliJ IDEA, is required. Java JDK 17 or above needs to be installed.

The code for this recipe is available in the `ch16/rest_plugin` directory.

How to do it...

In this recipe, we will see that a REST call is converted to an internal cluster action. To execute an internal cluster action, the following classes are required:

- A `Request` and `Response` class to communicate with the cluster.
- A `RequestBuilder` class is used to execute a request to the cluster.
- An `Action` class used to register the action and bound `Request`, `Response`, and `RequestBuilder`.

- A `Transport*Action` to bind the request and response to `ShardResponse`: it manages the *reduce* part of the query.
- A `ShardResponse` to manage the shard results.

We will perform the following steps:

1. We write a `SimpleRequest` class as follows:

```
public class SimpleRequest extends
BroadcastRequest<SimpleRequest> {
    private String field;
    SimpleRequest() {}
    public SimpleRequest(String... indices) {
        super(indices);
    }
    public SimpleRequest(StreamInput in) throws
IOException {
        super(in);
        field = in.readString();
    }
    public void setField(String field) {
        this.field = field;
    }
    public String getField() { return field; }
    @Override
    public void writeTo(StreamOutput out) throws
IOException {
        super.writeTo(out);
        out.writeString(field); } }
```

2. The `SimpleResponse` class is very similar to the `SimpleRequest` class.
3. To bind the request and the response, an action (`SimpleAction`) is required as follows:

```
public class SimpleAction extends
ActionType<SimpleResponse> {
    public static final SimpleAction INSTANCE = new
SimpleAction();
    public static final String NAME = "custom:indices/
simple";
```

```
    private SimpleAction() {
        super(NAME, SimpleResponse::new);
    }
}
```

4. The `Transport` class is the core of the action. It's quite long, so we'll only present the main important parts as follows:

```
public class TransportSimpleAction
        extends
TransportBroadcastByNodeAction<SimpleRequest,
SimpleResponse, ShardSimpleResponse> {
    private final IndicesService indicesService;
    @Inject
    public TransportSimpleAction(ClusterService
clusterService,
                                 TransportService
transportService, IndicesService indicesService,
                                 ActionFilters
actionFilters, IndexNameExpressionResolver
indexNameExpressionResolver) {
        super(SimpleAction.NAME, clusterService,
transportService, actionFilters,
                indexNameExpressionResolver,
SimpleRequest::new, ThreadPool.Names.SEARCH);
        this.indicesService = indicesService;
    }
    @Override
    protected SimpleResponse newResponse(SimpleRequest
request, int totalShards, int successfulShards, int
failedShards,

List<ShardSimpleResponse> shardSimpleResponses,

List<DefaultShardOperationFailedException> shardFailures,
                                         ClusterState
clusterState) {
        Set<String> simple = new HashSet<String>();
        for (ShardSimpleResponse shardSimpleResponse :
shardSimpleResponses) {
```

```
            simple.addAll(shardSimpleResponse.
getTermList());
        }
        return new SimpleResponse(totalShards,
successfulShards, failedShards, shardFailures, simple);
    }
    @Override
    protected void shardOperation(SimpleRequest request,
ShardRouting shardRouting, Task task,
            ActionListener<ShardSimpleResponse> listener)
{
        IndexService indexService = indicesService.
indexServiceSafe(shardRouting.shardId().getIndex());
        IndexShard indexShard = indexService.
getShard(shardRouting.shardId().id());
        indexShard.store().directory();
        Set<String> set = new HashSet<String>();
        set.add(request.getField() + "_" + shardRouting.
shardId());
        listener.onResponse(new
ShardSimpleResponse(shardRouting, set));
    }
    @Override
    protected ShardSimpleResponse
readShardResult(StreamInput in) throws IOException {
        return ShardSimpleResponse.readShardResult(in);
    }
    @Override
    protected SimpleRequest readRequestFrom(StreamInput
in) throws IOException {
        return new SimpleRequest(in);
    }
    @Override
    protected ShardsIterator shards(ClusterState
clusterState, SimpleRequest request, String[]
concreteIndices) {
        return clusterState.routingTable().
allShards(concreteIndices);
    }
```

```
    @Override
    protected ClusterBlockException
checkGlobalBlock(ClusterState state, SimpleRequest
request) {
        return state.blocks().
globalBlockedException(ClusterBlockLevel.METADATA_READ);
    }
    @Override
    protected ClusterBlockException
checkRequestBlock(ClusterState state, SimpleRequest
request, String[] concreteIndices) {
        return state.blocks().
indicesBlockedException(ClusterBlockLevel.METADATA_READ,
concreteIndices);
    }
}
```

How it works...

In this example, we used an action that is executed in every cluster node and for every shard that is selected on that node.

As you have seen, to execute a cluster action, the following classes are required:

- A couple of `Request/Response` class to interact with the cluster
- A task action on the cluster level
- A shard `Response` class to interact with the shards
- A `Transport` class to manage the map/reduce shard part that must be invoked by the REST call

These classes must extend one of the supported actions, for example:

- `TrasportBroadcastAction`: For actions that must be spread across the entire cluster.
- `TransportClusterInfoAction`: For actions that need to read information at the cluster level.
- `TransportMasterNodeAction`: For actions that must be executed only by the master node (such as index and mapping configuration). For a simple acknowledgment on the master, there is also the `AcknowledgedRequest` response.

- `TransportNodeAction`: For actions that must be executed on nodes (that is, all the node statistic actions).
- `TransportBroadcastReplicationAction`, `TransportReplicationAction`, `TransportWriteAction`: For actions that must be executed by a particular replica, first on primary and then on secondary ones.
- `TransportInstanceSingleOperationAction`: For actions that must be executed as a singleton in the cluster.
- `TransportSingleShardAction`: For actions that must be executed only in a shard (that is, `GET` actions). If it fails on a shard, it automatically tries on the shard replicas.
- `TransportTasksAction`: For actions that need to interact with cluster tasks.

In our example, we have defined an action that will be broadcast to every node and for every node, it collects its shard result and then it aggregates as follows:

```
public class TransportSimpleAction
        extends TransportBroadcastByNodeAction<SimpleRequest,
SimpleResponse, ShardSimpleResponse> {
```

All the request/response classes extend an `ActionResponse` class, so the following two methods for serializing their content must be provided:

- A contructor, which reads from a `StreamInput`, a class that encapsulates common input stream operations. This method allows the deserialization of the data we transmit on the wire. In the preceding example, we read a string with the following code:

```
public SimpleResponse(StreamInput in) throws IOException
{
    super(in);
    int n = in.readInt();
    simple = new HashSet<String>();
    for (int i = 0; i < n; i++) {
        simple.add(in.readString());
    }
}
```

- `writeTo`, which writes the contents of the class to be sent via the network. `StreamOutput` provides convenient methods to process the output. In the following example, we serialized the `StreamOutput` string:

```
@Override
public void writeTo(StreamOutput out) throws IOException
{
    super.writeTo(out);
    out.writeInt(simple.size());
    for (String t : simple) {
        out.writeString(t);
    }
}
```

In both actions, `super` must be called to allow the correct serialization of parent classes.

Every internal action in Elasticsearch is designed as a request/response pattern:

1. To complete the request/response action, we must define an action that binds the request with the correct response and a builder to construct it. To do so, we need to define an `Action` class as follows:

```
public class SimpleAction extends
ActionType<SimpleResponse> {
```

2. This `Action` object is a singleton object. We obtain it by creating a default static instance and private constructors as follows:

```
public static final SimpleAction INSTANCE = new
SimpleAction();
```

3. The `NAME` static string is used to uniquely identify the action at the cluster level.

```
public static final String NAME = "custom:indices/
simple";
```

4. To complete the `Action` definition, in the definition of the super constructor, we need to define a lambda to create a `Response`, which is used to create a new empty response as follows:

```
private SimpleAction() {
    super(NAME, SimpleResponse::new);
}
```

When the action is executed, the request and the response are serialized and sent to the cluster. To execute our custom code at the cluster level, a transport action is required.

The transport actions are usually defined as map and reduce jobs. The map part consists of executing the action on several shards and then reducing parts consisting of collecting all the results from the shards in a response that must be sent back to the requester. To speed up the process in Elasticsearch 5.x or above, all the shard's responses that belong in the same node are reduced in place to optimize the I/O and the network usage.

The transport action is a long class with many methods, but the most important ones are `ShardOperation` (map part) and `newResponse` (reduce part).

The original request is converted to a distributed `ShardRequest` that is processed by the `shardOperation` method as follows:

```
@Override
protected void shardOperation(SimpleRequest request,
ShardRouting shardRouting, Task task,
            ActionListener<ShardSimpleResponse> listener) {
```

To obtain the internal shard, we need to ask `IndexService` to return a shard based on the wanted index.

The shard request contains the index and the ID of the shard that must be used to execute the action as follows:

```
IndexService indexService = indicesService.
indexServiceSafe(shardRouting.shardId().getIndex());
IndexShard indexShard = indexService.getShard(shardRouting.
shardId().id());
```

The `IndexShard` object allows the execution of every possible shard operation (`search`, `get`, `index`, and many others). By means of this method, we can execute every data shard manipulation that we want.

> **Tip**
>
> Custom shard action can execute the application's business operation in a distributed and fast way.

1. In the following example, we have created a simple set of values:

```
indexShard.store().directory();
Set<String> set = new HashSet<String>();
```

```
set.add(request.getField() + "_" + shardRouting.
shardId());
```

2. The final step of our shard operation is to create a response to send back to the reduce step. In creating `ShardResponse`, we need to return the result plus information about the index and the shard that executed the action via `listener` as follows:

```
listener.onResponse(new ShardSimpleResponse(shardRouting,
set));
```

3. The distributed shard operations are collected in the reduce step (the `newResponse` method). This step aggregates all the shard results and sends back the result to the original `Action` as follows:

```
@Override
protected SimpleResponse newResponse(SimpleRequest
request, int totalShards, int successfulShards, int
failedShards,

List<ShardSimpleResponse> shardSimpleResponses,

List<DefaultShardOperationFailedException> shardFailures,
                                        ClusterState
clusterState) {
```

Other than the shard's result, the methods receive the status of the shard level operation and they are collected in three values: `successfulShards`, `failedShards`, and `shardFailures`.

4. The request result is a set of collected strings, so we create an empty set to collect the term's results as follows:

```
Set<String> simple = new HashSet<String>();
return new SimpleResponse(totalShards, successfulShards,
failedShards, shardFailures, simple);
```

5. Then you collect the results that we need to iterate on the shard responses as follows:

```
for (ShardSimpleResponse shardSimpleResponse :
shardSimpleResponses) {
    simple.addAll(shardSimpleResponse.getTermList());
}
```

6. The final step is to create the response by collecting the previous result and response status as follows:

```
return new SimpleResponse(totalShards, successfulShards,
failedShards, shardFailures, simple);
```

A cluster action needs to be created when there are low-level operations that we want to execute very quickly, such as special aggregations, server-side join, or a complex manipulation that requires several Elasticsearch calls to be executed. Writing custom Elasticsearch actions is an advanced Elasticsearch feature, but it can create new business use scenarios that can level up the capabilities of Elasticsearch.

See also

Refer to the *Creating a REST plugin* recipe in this chapter for how to interface the cluster action with a REST call.

Creating an ingest plugin

Elasticsearch 5.x introduced the ingest node that allows the modification, via a pipeline, to the records before ingesting in Elasticsearch. We have already seen in *Chapter 12*, *Using the Ingest Module*, that a pipeline is composed of one or more processor actions. In this recipe, we will see how to create a custom processor that stores in a field the initial character of another one.

Getting ready

You need an up and running Elasticsearch installation, as we described in the *Downloading and installing Elasticsearch* recipe in *Chapter 1*, *Getting Started*.

Gradle or an **integrated development environment** (**IDE**) that supports Java programming with Gradle (version 7.3.x used in the examples), such as Eclipse, Visual Studio Code, or IntelliJ IDEA, is required. Java JDK 17 or above needs to be installed.

The code for this recipe is available in the `ch16/ingest_plugin` directory.

How to do it...

To create an ingest processor plugin, we need to create the processor and then register it in the `plugin` class. We will perform the following steps:

1. We create the processor, and its factory, as follows:

   The class declaration and internal attributes:

```
public final class InitialProcessor extends
AbstractProcessor {
    public static final String TYPE = "initial";
    private final String field;
    private final String targetField;
    private final String defaultValue;
    private final boolean ignoreMissing;
    public InitialProcessor(String tag, String
description, String field, String targetField, boolean
ignoreMissing, String defaultValue) {
        super(tag, description);
        this.field = field;
        this.targetField = targetField;
        this.ignoreMissing = ignoreMissing;
        this.defaultValue = defaultValue;
    }
```

   The helper methods to access private attributes:

```
    String getField() { return field; }
    String getTargetField() { return targetField; }
    String getDefaultField() { return defaultValue; }
    boolean isIgnoreMissing() {return ignoreMissing;}
    @Override
    public String getType() { return TYPE; }
```

   The `execute` function, which is the core of the processors:

```
    @Override
    public IngestDocument execute(IngestDocument
document) {
        if (document.hasField(field, true) == false) {
            if (ignoreMissing) { return document;
            } else {
```

```
                throw new IllegalArgumentException("field
[" + field + "] not present as part of path [" + field +
"]");
            }
        }
        // We fail here if the target field point to an
array slot that is out of range.
        // If we didn't do this then we would fail if we
set the value in the target_field
        // and then on failure processors would not see
that value we tried to rename as we already
        // removed it.
        if (document.hasField(targetField, true)) {
            throw new IllegalArgumentException("field ["
+ targetField + "] already exists");
        }
        Object value = document.getFieldValue(field,
Object.class);
        if( value!=null && value instanceof String ) {
            String myValue=value.toString().trim();
            if(myValue.length()>1){
                try {
                    document.setFieldValue(targetField,
myValue.substring(0,1).toLowerCase(Locale.getDefault()));
                } catch (Exception e) {
                    // setting the value back to the
original field shouldn't as we just fetched it from that
field:
                    document.setFieldValue(field, value);
                    throw e;
                }
            }
        }
        return document;
    }
```

The `factory` used to initialize the processor:

```
    public static final class Factory implements
Processor.Factory {
        @Override
        public Processor create(Map<String, Processor.
Factory> processorFactories, String tag, String
description, Map<String, Object> config) throws Exception
{
            String field = ConfigurationUtils.
readStringProperty(TYPE, tag, config, "field");
            String targetField = ConfigurationUtils.
readStringProperty(TYPE, tag,
                    config, "target_field");
            String defaultValue = ConfigurationUtils.
readOptionalStringProperty(TYPE, tag,
                    config, "defaultValue");
            boolean ignoreMissing = ConfigurationUtils.
readBooleanProperty(TYPE, tag,
                    config, "ignore_missing", false);
            return new InitialProcessor(tag, description,
field, targetField, ignoreMissing, defaultValue);
        }
    }
}
```

2. We need to register it in the `Plugin` class with the following lines:

```
public class InitialIngestPlugin extends Plugin
implements IngestPlugin {
    @Override
    public Map<String, Processor.Factory>
getProcessors(Processor.Parameters parameters) {
        return Collections.singletonMap(InitialProcessor.
TYPE, new InitialProcessor.Factory());
    }}
```

3. Now we can build the plugin via `gradlew clean check` and manually install the ZIP. If we restart the Elasticsearch server, we should see the plugin loaded as follows:

```
… truncated …
[2021-05-01T20:55:39,740][INFO ][o.e.p.PluginsService
] [iMacParo] loaded module [x-pack-watcher]
[2021-05-01T20:55:39,741][INFO ][o.e.p.PluginsService
] [iMacParo] loaded plugin [initial-processor]
[2021-05-01T20:55:39,780][INFO ][o.e.e.NodeEnvironment
] [iMacParo] using [1] data paths, mounts [[/System/
Volumes/Data (/dev/disk1s1)]], net usable_space [16.9gb],
net total_space [931.6gb], types [apfs]
... truncated ...
```

4. We can test our custom ingest plugin via the Simulate Ingest API with a `curl` as follows:

```
curl -XPOST -H "Content-Type: application/
json" 'http://127.0.0.1:9200/_ingest/pipeline/_
simulate?verbose&pretty' -d '{
"pipeline": {
     "description": "Test my custom plugin",
     "processors": [
      {
         "initial": {
           "field": "user",
           "target_field": "user_initial"
        } } ], "version": 1 },
"docs": [
    { "_source": { "user": "john" } },
    { "_source": { "user": "Nancy" } } ] }'
```

5. The result will be something similar to the following:

```
{ "docs" : [
    { "processor_results" : [
        { "processor_type" : "initial",
          "status" : "success",
          "doc" : {
```

```
              "_index" : "_index", "_id" : "_id",
              "_source" : {
                "user_initial" : "j", "user" : "john" },
              "_ingest" : {
                "pipeline" : "_simulate_pipeline",
                "timestamp" : "2021-05-01T18:57:58.316032Z"
  } } } ] },
      { "processor_results" : [
          { "processor_type" : "initial",
            "status" : "success",
            "doc" : {
              "_index" : "_index", "_id" : "_id",
              "_source" : {
                "user_initial" : "n", "user" : "Nancy"},
              "_ingest" : {
                "pipeline" : "_simulate_pipeline",
                "timestamp" : "2021-05-01T18:57:58.316046Z"
  } } } ] } ] }
```

How it works...

First, you need to define the class that will manage your custom processor, which extends `AbstractProcessor`:

```
public final class InitialProcessor extends AbstractProcessor {
```

The `processor` class needs to know the fields on which it operates (`tag` name and `description` are mandatory). They are kept in the internal state of the processor as follows:

```
public InitialProcessor(String tag, String description, String
field, String targetField, boolean ignoreMissing, String
defaultValue) {
    super(tag, description);
    this.field = field;
    this.targetField = targetField;
    this.ignoreMissing = ignoreMissing;
    this.defaultValue = defaultValue;}
```

The core of the processor is the `execute` function, which contains our processor login as follows:

```
@Override
public IngestDocument execute(IngestDocument document) {
```

The `execute` function comprises the following steps:

1. Check whether the `source` field exists as follows:

```
if (!document.hasField(field, true)) {
    if (ignoreMissing) { return document; } else {
        throw new IllegalArgumentException("field [" +
field + "] not present as part of path [" + field + "]");
    } }
```

2. Check whether the `target` field does not exist as follows:

```
if (document.hasField(targetField, true)) {
    throw new IllegalArgumentException("field [" +
targetField + "] already exists"); }
```

3. We extract the value from `document` and check whether it's valid as follows:

```
Object value = document.getFieldValue(field, Object.
class);
if( value!=null && value instanceof String ) {
```

4. Now, we can process the value and set in the `target` field as follows:

```
String myValue=value.toString().trim();
if(myValue.length()>1){
    try {
        document.setFieldValue(targetField, myValue.
substring(0,1).toLowerCase(Locale.getDefault()));
    } catch (Exception e) {
        // setting the value back to the original field
shouldn't as we just fetched it from that field:
        document.setFieldValue(field, value);
        throw e;
    } }
return document;
```

5. To be able to initialize the processor for its definition, we need to define a `Factory` object as follows:

```
public static final class Factory implements Processor.
Factory {
```

6. The `Factory` object contains the `create` method that receives the registered processors, `processorTag`, and its configuration, which must be read as follows:

```
@Override
public Processor create(Map<String, Processor.Factory>
processorFactories, String tag, String description,
Map<String, Object> config) throws Exception {
    String field = ConfigurationUtils.
readStringProperty(TYPE, tag, config, "field");
    String targetField = ConfigurationUtils.
readStringProperty(TYPE, tag,
            config, "target_field");
    String defaultValue = ConfigurationUtils.
readOptionalStringProperty(TYPE, tag,
            config, "defaultValue");
    boolean ignoreMissing = ConfigurationUtils.
readBooleanProperty(TYPE, tag,
            config, "ignore_missing", false);
```

7. After having recovered, we can initialize the processor parameters as follows:

```
    return new InitialProcessor(tag, description, field,
targetField, ignoreMissing, defaultValue);
}
```

8. To be used as a custom processor, it needs to be registered in the plugin. This is done by extending the plugin as `IngestPlugin` as follows:

```
public class InitialIngestPlugin extends Plugin
implements IngestPlugin {
```

9. Now, we can register the `Factory` plugin in the `getProcessors` method as follows:

```
@Override
public Map<String, Processor.Factory>
getProcessors(Processor.Parameters parameters) {
```

```
    return Collections.singletonMap(InitialProcessor.
TYPE, new InitialProcessor.Factory());
}
```

Implementing an ingestion processor via a plugin is quite simple, and it's an incredibly powerful feature. With this approach, a user can create a custom step in enrichment pipelines.

See also

You can refer to the following URLs for further reference, which are related to this recipe:

- The official Elasticsearch documentation about the Ingest pipeline at `https://www.elastic.co/guide/en/elasticsearch/reference/current/ingest.html`, and *Chapter 12*, *Using the Ingest Module*
- The official Elasticsearch page of Ingest plugins at `https://www.elastic.co/guide/en/elasticsearch/plugins/current/ingest.html`