Contributing to elasticsearch

Elasticsearch is a free and open project and we love to receive contributions from our community — you! There are many ways to contribute, from writing tutorials or blog posts, improving the documentation, submitting bug reports and feature requests or writing code which can be incorporated into Elasticsearch itself.

If you want to be rewarded for your contributions, sign up for the Elastic Contributor Program. Each time you make a valid contribution, you'll earn points that increase your chances of winning prizes and being recognized as a top contributor.

Bug reports

If you think you have found a bug in Elasticsearch, first make sure that you are testing against the latest version of Elasticsearch - your issue may already have been fixed. If not, search our issues list on GitHub in case a similar issue has already been opened.

It is very helpful if you can prepare a reproduction of the bug. In other words, provide a small test case which we can run to confirm your bug. It makes it easier to find the problem and to fix it. Test cases should be provided as `curl` commands which we can copy and paste into a terminal to run it locally, for example:

```
# delete the index
curl -XDELETE localhost:9200/test

# insert a document
curl -XPUT localhost:9200/test/test/1 -d '{
 "title": "test document"
}'

# this should return XXXX but instead returns YYY
curl ....
```

Provide as much information as you can. You may think that the problem lies with your query, when actually it depends on how your data is indexed. The easier it is for us to recreate your problem, the faster it is likely to be fixed.

Feature requests

If you find yourself wishing for a feature that doesn't exist in Elasticsearch, you are probably not alone. There are bound to be others out there with similar needs. Many of the features that Elasticsearch has today have been added because our users saw the need. Open an issue on our issues list on GitHub

which describes the feature you would like to see, why you need it, and how it should work.

Contributing code and documentation changes

If you would like to contribute a new feature or a bug fix to Elasticsearch, please discuss your idea first on the GitHub issue. If there is no GitHub issue for your idea, please open one. It may be that somebody is already working on it, or that there are particular complexities that you should know about before starting the implementation. There are often a number of ways to fix a problem and it is important to find the right approach before spending time on a PR that cannot be merged.

We add the `help wanted` label to existing GitHub issues for which community contributions are particularly welcome, and we use the `good first issue` label to mark issues that we think will be suitable for new contributors.

The process for contributing to any of the Elastic repositories is similar. Details for individual projects can be found below.

Fork and clone the repository

You will need to fork the main Elasticsearch code or documentation repository and clone it to your local machine. See github help page for help.

Further instructions for specific projects are given below.

Tips for code changes

Following these tips prior to raising a pull request will speed up the review cycle.

- Add appropriate unit tests (details on writing tests can be found in the TESTING file)
- Add integration tests, if applicable
- Make sure the code you add follows the formatting guidelines
- Lines that are not part of your change should not be edited (e.g. don't format unchanged lines, don't reorder existing imports)
- Add the appropriate license headers to any new files
- For contributions involving the elasticsearch build you can find (details about the build setup in the
- BUILDING file

Submitting your changes

Once your changes and tests are ready to submit for review:

1. Test your changes

   Run the test suite to make sure that nothing is broken. See the TESTING file for help running tests.

2. Sign the Contributor License Agreement

   Please make sure you have signed our Contributor License Agreement. We are not asking you to assign copyright to us, but to give us the right to distribute your code without restriction. We ask this of all contributors in order to assure our users of the origin and continuing existence of the code. You only need to sign the CLA once.

3. Rebase your changes

   Update your local repository with the most recent code from the main Elasticsearch repository, and rebase your branch on top of the latest master branch. We prefer your initial changes to be squashed into a single commit. Later, if we ask you to make changes, add them as separate commits. This makes them easier to review. As a final step before merging we will either ask you to squash all commits yourself or we'll do it for you.

4. Submit a pull request

   Push your local changes to your forked copy of the repository and submit a pull request. In the pull request, choose a title which sums up the changes that you have made, and in the body provide more details about what your changes do. Also mention the number of the issue where discussion has taken place, eg "Closes #123".

Then sit back and wait. There will probably be discussion about the pull request and, if any changes are needed, we would love to work with you to get your pull request merged into Elasticsearch.

Please adhere to the general guideline that you should never force push to a publicly shared branch. Once you have opened your pull request, you should consider your branch publicly shared. Instead of force pushing you can just add incremental commits; this is generally easier on your reviewers. If you need to pick up changes from master, you can merge master into your branch. A reviewer might ask you to rebase a long-running pull request in which case force pushing is okay for that request. Note that squashing at the end of the review process should also not be done, that can be done when the pull request is integrated via GitHub.

Contributing to the Elasticsearch codebase

Repository: https://github.com/elastic/elasticsearch

JDK 17 is required to build Elasticsearch. You must have a JDK 17 installation with the environment variable `JAVA_HOME` referencing the path to Java home for your JDK 17 installation. By default, tests use the same runtime as `JAVA_HOME`. However, since Elasticsearch supports JDK 11, the build supports compiling with JDK 17 and testing on a JDK 11 runtime; to do this, set `RUNTIME_JAVA_HOME` pointing to the Java home of a JDK 11 installation. Note that this mechanism

can be used to test against other JDKs as well, this is not only limited to JDK 11.

> Note: It is also required to have `JAVA8_HOME`, `JAVA11_HOME`, and `JAVA17_HOME` available so that the tests can pass.

Elasticsearch uses the Gradle wrapper for its build. You can execute Gradle using the wrapper via the `gradlew` script on Unix systems or `gradlew.bat` script on Windows in the root of the repository. The examples below show the usage on Unix.

We support development in IntelliJ versions IntelliJ 2020.1 and onwards.

Docker is required for building some Elasticsearch artifacts and executing certain test suites. You can run Elasticsearch without building all the artifacts with:

```
./gradlew :run
```

That'll spend a while building Elasticsearch and then it'll start Elasticsearch, writing its log above Gradle's status message. We log a lot of stuff on startup, specifically these lines tell you that Elasticsearch is ready:

```
[2020-05-29T14:50:35,167][INFO ][o.e.h.AbstractHttpServerTransport] [runTask-0] publish_addr
[2020-05-29T14:50:35,169][INFO ][o.e.n.Node               ] [runTask-0] started
```

But to be honest its typically easier to wait until the console stops scrolling and then run `curl` in another window like this:

```
curl -u elastic:password localhost:9200
```

Importing the project into IntelliJ IDEA

The minimum IntelliJ IDEA version required to import the Elasticsearch project is 2020.1 Elasticsearch builds using Java 17. When importing into IntelliJ you will need to define an appropriate SDK. The convention is that **this SDK should be named "17"** so that the project import will detect it automatically. For more details on defining an SDK in IntelliJ please refer to their documentation. SDK definitions are global, so you can add the JDK from any project, or after project import. Importing with a missing JDK will still work, IntelliJ will simply report a problem and will refuse to build until resolved.

You can import the Elasticsearch project into IntelliJ IDEA via:

- Select **File > Open**
- In the subsequent dialog navigate to the root `build.gradle` file
- In the subsequent dialog select **Open as Project**

Checkstyle If you have the Checkstyle plugin installed, you can configure IntelliJ to check the Elasticsearch code. However, the Checkstyle configuration file does not work by default with the IntelliJ plugin, so instead an IDE-specific config file is generated automatically after IntelliJ finishes syncing. You can

manually generate the file with `./gradlew configureIdeCheckstyle` in case it is removed due to a `./gradlew clean` or other action.

IntelliJ should be automatically configured to use the generated rules after import via the `.idea/checkstyle-idea.xml` configuration file. No further action is required.

Formatting Elasticsearch code is automatically formatted with spotless, backed by the Eclipse formatter. You can do the same in IntelliJ with the Eclipse Code Formatter so that you can apply the correct formatting directly in your IDE. The configuration for the plugin is held in `.idea/eclipseCodeFormatter.xml` and should be automatically applied, but manual instructions are below in case you need them.

1. Open **Preferences > Other Settings > Eclipse Code Formatter**
2. Click "Use the Eclipse Code Formatter"
3. Under "Eclipse formatter config", select "Eclipse workspace/project folder or config file"
4. Click "Browse", and navigate to the file `build-conventions/formatterConfig.xml`
5. **IMPORTANT** - make sure "Optimize Imports" is **NOT** selected.
6. Click "OK"

Alternative manual steps for IntelliJ.

1. Open **File > Settings/Preferences > Code Style > Java**
2. Gear icon > Import Scheme > Eclipse XML Profile
3. Navigate to the file `build-conventions/formatterConfig.xml`
4. Click "OK"

REST Endpoint Conventions

Elasticsearch typically uses singular nouns rather than plurals in URLs. For example:

```
/_ingest/pipeline
/_ingest/pipeline/{id}
```

but not:

```
/_ingest/pipelines
/_ingest/pipelines/{id}
```

You may find counterexamples, but new endpoints should use the singular form.

Java Language Formatting Guidelines

Java files in the Elasticsearch codebase are automatically formatted using the Spotless Gradle plugin. All new projects are automatically formatted, while existing projects are gradually being opted-in. The formatting check is run automatically via the `precommit` task, but it can be run explicitly with:

```
./gradlew spotlessJavaCheck
```

It is usually more useful, and just as fast, to just reformat the project. You can do this with:

```
./gradlew spotlessApply
```

These tasks can also be run for specific subprojects, e.g.

```
./gradlew server:spotlessJavaCheck
```

Please follow these formatting guidelines:

- Java indent is 4 spaces
- Line width is 140 characters
- Lines of code surrounded by `// tag::NAME` and `// end::NAME` comments are included in the documentation and should only be 76 characters wide not counting leading indentation. Such regions of code are not formatted automatically as it is not possible to change the line length rule of the formatter for part of a file. Please format such sections sympathetically with the rest of the code, while keeping lines to maximum length of 76 characters.
- Wildcard imports (`import foo.bar.baz.*`) are forbidden and will cause the build to fail.
- If *absolutely* necessary, you can disable formatting for regions of code with the `// tag::noformat` and `// end::noformat` directives, but only do this where the benefit clearly outweighs the decrease in formatting consistency.
- Note that Javadoc and block comments i.e. `/* ... */` are not formatted, but line comments i.e `// ...` are.
- Negative boolean expressions must use the form `foo == false` instead of `!foo` for better readability of the code. This is enforced via Checkstyle. Conversely, you should not write e.g. `if (foo == true)`, but just `if (foo)`.

Editor / IDE Support IntelliJ IDEs can import the same settings file, and / or use the Eclipse Code Formatter plugin.

You can also tell Spotless to format a specific file from the command line.

Javadoc

Good Javadoc can help with navigating and understanding code. Elasticsearch has some guidelines around when to write Javadoc and when not to, but note that we don't want to be overly prescriptive. The intent of these guidelines is to be helpful, not to turn writing code into a chore.

The short version

1. Always add Javadoc to new code.
2. Add Javadoc to existing code if you can.

3. Document the "why", not the "how", unless that's important to the "why".
4. Don't document anything trivial or obvious (e.g. getters and setters). In other words, the Javadoc should add some value.

The long version

1. If you add a new Java package, please also add package-level Javadoc that explains what the package is for. This can just be a reference to a more foundational / parent package if appropriate. An example would be a package hierarchy for a new feature or plugin - the package docs could explain the purpose of the feature, any caveats, and possibly some examples of configuration and usage.
2. New classes and interfaces must have class-level Javadoc that describes their purpose. There are a lot of classes in the Elasticsearch repository, and it's easier to navigate when you can quickly find out what is the purpose of a class. This doesn't apply to inner classes or interfaces, unless you expect them to be explicitly used outside their parent class.
3. New public methods must have Javadoc, because they form part of the contract between the class and its consumers. Similarly, new abstract methods must have Javadoc because they are part of the contract between a class and its subclasses. It's important that contributors know why they need to implement a method, and the Javadoc should make this clear. You don't need to document a method if it's overriding an abstract method (either from an abstract superclass or an interface), unless your implementation is doing something "unexpected" e.g. deviating from the intent of the original method.
4. Following on from the above point, please add docs to existing public methods if you are editing them, or to abstract methods if you can.
5. Non-public, non-abstract methods don't require Javadoc, but if you feel that adding some would make it easier for other developers to understand the code, or why it's written in a particular way, then please do so.
6. Properties don't need to have Javadoc, but please add some if there's something useful to say.
7. Javadoc should not go into low-level implementation details unless this is critical to understanding the code e.g. documenting the subtleties of the implementation of a private method. The point here is that implementations will change over time, and the Javadoc is less likely to become out-of-date if it only talks about the what is the purpose of the code, not what it does.
8. Examples in Javadoc can be very useful, so feel free to add some if you can reasonably do so i.e. if it takes a whole page of code to set up an example, then Javadoc probably isn't the right place for it. Longer or more elaborate examples are probably better suited to the package docs.
9. Test methods are a good place to add Javadoc, because you can use it to succinctly describe e.g. preconditions, actions and expectations of the test, more easily that just using the test name alone. Please consider

documenting your tests in this way.

10. Sometimes you shouldn't add Javadoc:
    1. Where it adds no value, for example where a method's implementation is trivial such as with getters and setters, or a method just delegates to another object.
    2. However, you should still add Javadoc if there are caveats around calling a method that are not immediately obvious from reading the method's implementation in isolation.
    3. You can omit Javadoc for simple classes, e.g. where they are a simple container for some data. However, please consider whether a reader might still benefit from some additional background, for example about why the class exists at all.
11. Not all comments need to be Javadoc. Sometimes it will make more sense to add comments in a method's body, for example due to important implementation decisions or "gotchas". As a general guide, if some information forms part of the contract between a method and its callers, then it should go in the Javadoc, otherwise you might consider using regular comments in the code. Remember as well that Elasticsearch has extensive user documentation, and it is not the role of Javadoc to replace that.
    - If a method's performance is "unexpected" then it's good to call that out in the Javadoc. This is especially helpful if the method is usually fast but sometimes very slow (shakes fist at caching).
12. Please still try to make class, method or variable names as descriptive and concise as possible, as opposed to relying solely on Javadoc to describe something.
13. Use `@link` to add references to related resources in the codebase. Or outside the code base.
    1. `@see` is much more limited than `@link`. You can use it but most of the time `@link` flows better.
14. If you need help writing Javadoc, just ask!

Finally, use your judgement! Base your decisions on what will help other developers - including yourself, when you come back to some code 3 months in the future, having forgotten how it works.

License Headers

We require license headers on all Java files. With the exception of the top-level `x-pack` directory, all contributed code should have the following license header unless instructed otherwise:

```
/*
 * Copyright Elasticsearch B.V. and/or licensed to Elasticsearch B.V. under one
 * or more contributor license agreements. Licensed under the Elastic License
 * 2.0 and the Server Side Public License, v 1; you may not use this file except
 * in compliance with, at your election, the Elastic License 2.0 or the Server
 * Side Public License, v 1.
```

```
 */
```

The top-level `x-pack` directory contains code covered by the Elastic license. Community contributions to this code are welcome, and should have the following license header unless instructed otherwise:

```
/*
 * Copyright Elasticsearch B.V. and/or licensed to Elasticsearch B.V. under one
 * or more contributor license agreements. Licensed under the Elastic License
 * 2.0; you may not use this file except in compliance with the Elastic License
 * 2.0.
 */
```

It is important that the only code covered by the Elastic licence is contained within the top-level `x-pack` directory. The build will fail its pre-commit checks if contributed code does not have the appropriate license headers.

> **NOTE:** If you have imported the project into IntelliJ IDEA the project will be automatically configured to add the correct license header to new source files based on the source location.

Type-checking, generics and casting

You should try to write code that does not require suppressing any warnings from the compiler, e.g. suppressing type-checking, raw generics, and so on. However, this isn't always possible or practical. In such cases, you should use the `@SuppressWarnings` annotations to silence the compiler warning, trying to keep the scope of the suppression as small as possible. Where a piece of code requires a lot of suppressions, it may be better to apply a single suppression at a higher level e.g. at the method or even class level. Use your judgement.

There are also cases where the compiler simply refuses to accept an assignment or cast of any kind, because it lacks the information to know that the types are OK. In such cases, you can use the `Types.forciblyCast` utility method. As the name suggests, you can coerce any type to any other type, so please use it as a last resort.

Logging

The Elasticsearch server logs are vitally useful for diagnosing problems in a running cluster. You should make sure that your contribution uses logging appropriately: log enough detail to inform users about key events and help them understand what happened when things go wrong without logging so much detail that the logs fill up with noise and the useful signal is lost.

Elasticsearch uses Log4J for logging. In most cases you should log via a `Logger` named after the class that is writing the log messages, which you can do by declaring a static field of the class. For example:

```
class Foo {
```

```
    private static final Logger logger = LogManager.getLogger(Foo.class);
}
```

In rare situations you may want to configure your `Logger` slightly differently, perhaps specifying a different class or maybe using one of the methods on `org.elasticsearch.common.logging.Loggers` instead.

If the log message includes values from your code then you must use use placeholders rather than constructing the string yourself using simple concatenation. Consider wrapping the values in `[...]` to help distinguish them from the static part of the message:

```
logger.debug("operation failed [{}] times in [{}]ms", failureCount, elapsedMillis);
```

You can also pass in an exception to log it including its stack trace, and any causes and their causes, as well as any suppressed exceptions and so on:

```
logger.debug("operation failed", exception);
```

If you wish to use placeholders and an exception at the same time, construct a `ParameterizedMessage`:

```
logger.debug(new ParameterizedMessage("failed at offset [{}]", offset), exception);
```

You can also use a `Supplier<ParameterizedMessage>` to avoid constructing expensive messages that will usually be discarded:

```
logger.debug(() -> new ParameterizedMessage("rarely seen output [{}]", expensiveMethod()));
```

Logging is an important behaviour of the system and sometimes deserves its own unit tests, especially if there is complex logic for computing what is logged and when to log it. You can use a `org.elasticsearch.test.MockLogAppender` to make assertions about the logs that are being emitted.

Logging is a powerful diagnostic technique but it is not the only possibility. You should also consider exposing some information about your component via an API instead of in logs. For instance you can implement APIs to report its current status, various statistics, and maybe even details of recent failures.

Log levels Each log message is written at a particular *level*. By default Elasticsearch will suppress messages at the two most verbose levels, `TRACE` and `DEBUG`, and will output messages at all other levels. Users can configure which levels of message are written by each logger at runtime, but you should expect everyone to run with the default configuration almost all of the time and choose your levels accordingly.

The guidance in this section is subjective in some areas. When in doubt, discuss your choices with reviewers.

`TRACE` This is the most verbose level, disabled by default, and it is acceptable if it generates a very high volume of logs. The target audience of `TRACE` logs comprises

developers who are trying to deeply understand some unusual runtime behaviour of a system. For instance `TRACE` logs may be useful when understanding an unexpected interleaving of concurrent actions or some unexpected consequences of a delayed response from a remote node.

`TRACE` logs will normally only make sense when read alongside the code, and typically they will be read as a whole sequence of messages rather than in isolation. For example, the `InternalClusterInfoService` uses `TRACE` logs to record certain key events in its periodic refresh process:

```
logger.trace("starting async refresh");
// ...
logger.trace("received node stats response");
// ...
logger.trace("received indices stats response");
// ...
logger.trace("stats all received, computing cluster info and notifying listeners");
// ...
logger.trace("notifying [{}] of new cluster info", listener);
```

Even though `TRACE` logs may be very verbose, you should still exercise some judgement when deciding when to use them. In many cases it will be easier to understand the behaviour of the system using tests or by analysing the code itself rather than by trawling through hundreds of trivial log messages.

It may not be easy, or even possible, to obtain `TRACE` logs from a production system. Therefore they are not appropriate for information that you would normally expect to be useful in diagnosing problems in production.

`DEBUG` This is the next least verbose level and is also disabled by default. The target audience of this level typically comprises users or developers who are trying to diagnose an unexpected problem in a production system, perhaps to help determine whether a fault lies within Elasticsearch or elsewhere.

Users should expect to be able to enable `DEBUG` logging on their production systems for a whole subsystem for an extended period of time without overwhelming the system or filling up their disks with logs, so it is important to limit the volume of messages logged at this level. On the other hand, these messages must still provide enough detail to diagnose the sorts of problems that you expect Elasticsearch to encounter. In some cases it works well to collect information over a period of time and then log a complete summary, rather than recording every step of a process in its own message.

For example, the `Coordinator` uses `DEBUG` logs to record a change in mode, including various internal details for context, because this event is fairly rare but not important enough to notify users by default:

```
logger.debug(
    "{}: coordinator becoming CANDIDATE in term {} (was {}, lastKnownLeader was [{}])",
```

```
    method,
    getCurrentTerm(),
    mode,
    lastKnownLeader
);
```

It's possible that the reader of `DEBUG` logs is also reading the code, but that is less likely than for `TRACE` logs. Strive to avoid terminology that only makes sense when reading the code, and also aim for messages at this level to be self-contained rather than intending them to be read as a sequence.

It's often useful to log exceptions and other deviations from the "happy path" at `DEBUG` level. Exceptions logged at `DEBUG` should generally include the complete stack trace.

`INFO` This is the next least verbose level, and the first level that is enabled by default. It is appropriate for recording important events in the life of the cluster, such as an index being created or deleted or a snapshot starting or completing. Users will mostly ignore log messages at `INFO` level, but may use these messages to construct a high-level timeline of events leading up to an incident.

For example, the `MetadataIndexTemplateService` uses `INFO` logs to record when an index template is created or updated:

```
logger.info(
    "{} index template [{}] for index patterns {}",
    existing == null ? "adding" : "updating",
    name,
    template.indexPatterns()
);
```

`INFO`-level logging is enabled by default so its target audience is the general population of users and administrators. You should use user-facing terminology and ensure that messages at this level are self-contained. In general you shouldn't log unusual events, particularly exceptions with stack traces, at `INFO` level. If the event is relatively benign then use `DEBUG`, whereas if the user should be notified then use `WARN`.

Bear in mind that users will be reading the logs when they're trying to determine why their node is not behaving the way they expect. If a log message sounds like an error then some users will interpret it as one, even if it is logged at `INFO` level. Where possible, `INFO` messages should prefer factual over judgemental language, for instance saying `Did not find ...` rather than `Failed to find ...`.

`WARN` This is the next least verbose level, and is also enabled by default. Ideally a healthy cluster will emit no `WARN`-level logs, but this is the appropriate level for recording events that the cluster administrator should investigate, or which indicate a bug. Some production environments require the cluster to emit no

`WARN`-level logs during acceptance testing, so you must ensure that any logs at this level really do indicate a problem that needs addressing.

As with the `INFO` level, you should use user-facing terminology at the `WARN` level, and also ensure that messages are self-contained. Strive to make them actionable too since you should be logging at this level when the user should take some investigative action.

For example, the `DiskThresholdMonitor` uses `WARN` logs to record that a disk threshold has been breached:

```
logger.warn(
    "flood stage disk watermark [{}] exceeded on {}, all indices on this node will be marked
    diskThresholdSettings.describeFloodStageThreshold(),
    usage
);
```

Unlike at the `INFO` level, it is often appropriate to log an exception, complete with stack trace, at `WARN` level. Although the stack trace may not be useful to the user, it may contain information that is vital for a developer to fully understand the problem and its wider context.

In a situation where occasional transient failures are expected and handled, but a persistent failure requires the user's attention, consider implementing a mechanism to detect that a failure is unacceptably persistent and emit a corresponding `WARN` log. For example, it may be helpful to log every tenth consecutive failure at `WARN` level, or log at `WARN` if an operation has not completed within a certain time limit. This is much more user-friendly than failing persistently and silently by default and requiring the user to enable `DEBUG` logging to investigate the problem.

If an exception occurs as a direct result of a request received from a client then it should only be logged as a `WARN` if the server administrator is the right person to address it. In most cases the server administrator cannot do anything about faulty client requests, and the person running the client is often unable to see the server logs, so you should include the exception in the response back to the client and not log a warning. Bear in mind that clients may submit requests at a high rate, so any per-request logging can easily flood the logs.

`ERROR` This is the next least verbose level after `WARN`. In theory it is possible for users to suppress messages at `WARN` and below, believing this to help them focus on the most important `ERROR` messages, but in practice in Elasticsearch this will hide so much useful information that the resulting logs will be useless, so we do not expect users to do this kind of filtering.

On the other hand, users may be familiar with the `ERROR` level from elsewhere. Log4J for instance documents this level as meaning "an error in the application, possibly recoverable". The implication here is that the error is possibly *not*

recoverable too, and we do encounter users that get very worried by logs at `ERROR` level for this reason.

Therefore you should try and avoid logging at `ERROR` level unless the error really does indicate that Elasticsearch is now running in a degraded state from which it will not recover. For instance, the `FsHealthService` uses `ERROR` logs to record that the data path failed some basic health checks and hence the node cannot continue to operate as a member of the cluster:

```
logger.error(new ParameterizedMessage("health check of [{}] failed", path), ex);
```

Errors like this should be very rare. When in doubt, prefer `WARN` to `ERROR`.

Creating A Distribution

Run all build commands from within the root directory:

```
cd elasticsearch/
```

To build a darwin-tar distribution, run this command:

```
./gradlew -p distribution/archives/darwin-tar assemble
```

You will find the distribution under:

```
./distribution/archives/darwin-tar/build/distributions/
```

To create all build artifacts (e.g., plugins and Javadocs) as well as distributions in all formats, run this command:

```
./gradlew assemble
```

> **NOTE:** Running the task above will fail if you don't have an available Docker installation.

The package distributions (Debian and RPM) can be found under:

```
./distribution/packages/(deb|rpm|oss-deb|oss-rpm)/build/distributions/
```

The archive distributions (tar and zip) can be found under:

```
./distribution/archives/(darwin-tar|linux-tar|windows-zip|oss-darwin-tar|oss-linux-tar|oss-w
```

Running The Full Test Suite

Before submitting your changes, run the test suite to make sure that nothing is broken, with:

```
./gradlew check
```

If your changes affect only the documentation, run:

```
./gradlew -p docs check
```

For more information about testing code examples in the documentation, see https://github.com/elastic/elasticsearch/blob/master/docs/README.asciidoc

Project layout

This repository is split into many top level directories. The most important ones are:

`docs` Documentation for the project.

`distribution` Builds our tar and zip archives and our rpm and deb packages.

`libs` Libraries used to build other parts of the project. These are meant to be internal rather than general purpose. We have no plans to semver their APIs or accept feature requests for them. We publish them to maven central because they are dependencies of our plugin test framework, high level rest client, and jdbc driver but they really aren't general purpose enough to *belong* in maven central. We're still working out what to do here.

`modules` Features that are shipped with Elasticsearch by default but are not built in to the server. We typically separate features from the server because they require permissions that we don't believe *all* of Elasticsearch should have or because they depend on libraries that we don't believe *all* of Elasticsearch should depend on.

For example, reindex requires the `connect` permission so it can perform reindex-from-remote but we don't believe that the *all* of Elasticsearch should have the "connect". For another example, Painless is implemented using antlr4 and asm and we don't believe that *all* of Elasticsearch should have access to them.

`plugins` Officially supported plugins to Elasticsearch. We decide that a feature should be a plugin rather than shipped as a module because we feel that it is only important to a subset of users, especially if it requires extra dependencies.

The canonical example of this is the ICU analysis plugin. It is important for folks who want the fairly language neutral ICU analyzer but the library to implement the analyzer is 11MB so we don't ship it with Elasticsearch by default.

Another example is the `discovery-gce` plugin. It is *vital* to folks running in GCP but useless otherwise and it depends on a dozen extra jars.

`qa` Honestly this is kind of in flux and we're not 100% sure where we'll end up. Right now the directory contains * Tests that require multiple modules or plugins to work * Tests that form a cluster made up of multiple versions of Elasticsearch like full cluster restart, rolling restarts, and mixed version tests * Tests that test the Elasticsearch clients in "interesting" places like the `wildfly` project. * Tests that test Elasticsearch in funny configurations like with ingest disabled * Tests that need to do strange things like install plugins that thrown uncaught `Throwable`s or add a shutdown hook But we're not convinced that all of these things *belong* in the qa directory. We're fairly sure that tests that

require multiple modules or plugins to work should just pick a "home" plugin. We're fairly sure that the multi-version tests *do* belong in qa. Beyond that, we're not sure. If you want to add a new qa project, open a PR and be ready to discuss options.

`server` The server component of Elasticsearch that contains all of the modules and plugins. Right now things like the high level rest client depend on the server but we'd like to fix that in the future.

`test` Our test framework and test fixtures. We use the test framework for testing the server, the plugins, and modules, and pretty much everything else. We publish the test framework so folks who develop Elasticsearch plugins can use it to test the plugins. The test fixtures are external processes that we start before running specific tests that rely on them.

For example, we have an hdfs test that uses mini-hdfs to test our repository-hdfs plugin.

`x-pack` Commercially licensed code that integrates with the rest of Elasticsearch. The `docs` subdirectory functions just like the top level `docs` subdirectory and the `qa` subdirectory functions just like the top level `qa` subdirectory. The `plugin` subdirectory contains the x-pack module which runs inside the Elasticsearch process.

Gradle Build

We use Gradle to build Elasticsearch because it is flexible enough to not only build and package Elasticsearch, but also orchestrate all of the ways that we have to test Elasticsearch.

Configurations Gradle organizes dependencies and build artifacts into "configurations" and allows you to use these configurations arbitrarily. Here are some of the most common configurations in our build and how we use them:

`implementation`

Dependencies that are used by the project at compile and runtime but are not exposed as a compile dependency to other dependent projects. Dependencies added to the `implementation` configuration are considered an implementation detail that can be changed at a later date without affecting any dependent projects.

`api`

Dependencies that are used as compile and runtime dependencies of a project and are considered part of the external api of the project.

`runtimeOnly`

Dependencies that not on the classpath at compile time but are on the classpath at runtime. We mostly use this configuration to make sure that we do not accidentally compile against dependencies of our dependencies also known as "transitive" dependencies".

`compileOnly`

Code that is on the classpath at compile time but that should not be shipped with the project because it is "provided" by the runtime somehow. Elasticsearch plugins use this configuration to include dependencies that are bundled with Elasticsearch's server.

`testImplementation`

Code that is on the classpath for compiling tests that are part of this project but not production code. The canonical example of this is `junit`.

Reviewing and accepting your contribution

We review every contribution carefully to ensure that the change is of high quality and fits well with the rest of the Elasticsearch codebase. If accepted, we will merge your change and usually take care of backporting it to appropriate branches ourselves.

We really appreciate everyone who is interested in contributing to Elasticsearch and regret that we sometimes have to reject contributions even when they might appear to make genuine improvements to the system. Reviewing contributions can be a very time-consuming task, yet the team is small and our time is very limited. In some cases the time we would need to spend on reviews would outweigh the benefits of a change by preventing us from working on other more beneficial changes instead.

Please discuss your change in a Github issue before spending much time on its implementation. We sometimes have to reject contributions that duplicate other efforts, take the wrong approach to solving a problem, or solve a problem which does not need solving. An up-front discussion often saves a good deal of wasted time in these cases.

We normally immediately reject isolated PRs that only perform simple refactorings or otherwise "tidy up" certain aspects of the code. We think the benefits of this kind of change are very small, and in our experience it is not worth investing the substantial effort needed to review them. This especially includes changes suggested by tools.

We sometimes reject contributions due to the low quality of the submission since low-quality submissions tend to take unreasonable effort to review properly. Quality is rather subjective so it is hard to describe exactly how to avoid this, but there are some basic steps you can take to reduce the chances of rejection. Follow the guidelines listed above when preparing your changes. You should add tests that correspond with your changes, and your PR should pass affected test

suites too. It makes it much easier to review if your code is formatted correctly and does not include unnecessary extra changes.

We sometimes reject contributions if we find ourselves performing many review iterations without making enough progress. Some iteration is expected, particularly on technically complicated changes, and there's no fixed limit on the acceptable number of review cycles since it depends so much on the nature of the change. You can help to reduce the number of iterations by reviewing your contribution yourself or in your own team before asking us for a review. You may be surprised how many comments you can anticipate and address by taking a short break and then carefully looking over your changes again.

We expect you to follow up on review comments somewhat promptly, but recognise that everyone has many priorities for their time and may not be able to respond for several days. We will understand if you find yourself without the time to complete your contribution, but please let us know that you have stopped working on it. We will try to send you a reminder if we haven't heard from you in a while, but may end up closing your PR if you do not respond for too long.

If your contribution is rejected we will close the pull request with a comment explaining why. This decision isn't always final: if you feel we have misunderstood your intended change or otherwise think that we should reconsider then please continue the conversation with a comment on the pull request and we'll do our best to address any further points you raise.

Contributing as part of a class

In general Elasticsearch is happy to accept contributions that were created as part of a class but strongly advise against making the contribution as part of the class. So if you have code you wrote for a class feel free to submit it.

Please, please, please do not assign contributing to Elasticsearch as part of a class. If you really want to assign writing code for Elasticsearch as an assignment then the code contributions should be made to your private clone and opening PRs against the primary Elasticsearch clone must be optional, fully voluntary, not for a grade, and without any deadlines.

Because:

- While the code review process is likely very educational, it can take wildly varying amounts of time depending on who is available, where the change is, and how deep the change is. There is no way to predict how long it will take unless we rush.
- We do not rush reviews without a very, very good reason. Class deadlines aren't a good enough reason for us to rush reviews.
- We deeply discourage opening a PR you don't intend to work through the entire code review process because it wastes our time.

- We don't have the capacity to absorb an entire class full of new contributors, especially when they are unlikely to become long time contributors.

Finally, we require that you run `./gradlew check` before submitting a non-documentation contribution. This is mentioned above, but it is worth repeating in this section because it has come up in this context.