Building Elasticsearch with Gradle

Elasticsearch is built using the Gradle open source build tools.

This document provides a general guidelines for using and working on the elasticsearch build logic.

Build logic organisation

The Elasticsearch project contains 3 build-related projects that are included into the Elasticsearch build as a composite build.

`build-conventions`

This project contains build conventions that are applied to all elasticsearch projects.

`build-tools`

This project contains all build logic that we publish for third party elasticsearch plugin authors. We provide the following plugins:

- `elasticsearch.esplugin` - A gradle plugin for building an elasticsearch plugin.
- `elasticsearch.testclusters` - A gradle plugin for setting up es clusters for testing within a build.

This project is published as part of the elasticsearch release and accessible by `org.elasticsearch.gradle:build-tools:<versionNumber>`. These build tools are also used by the `elasticsearch-hadoop` project maintained by elastic.

`build-tools-internal`

This project contains all elasticsearch project specific build logic that is not meant to be shared with other internal or external projects.

Build guidelines

This is an intentionally small set of guidelines to build users and authors to ensure we keep the build consistent. We also publish elasticsearch build logic as `build-tools` to be usuable by thirdparty elasticsearch plugin authors. This is also used by other elastic teams like `elasticsearch-hadoop`. Breaking changes should therefore be avoided and an appropriate deprecation cycle should be followed.

Stay up to date

The elasticsearch build usually uses the latest Gradle GA release. We stay as close to the latest Gradle releases as possible. In certain cases an update is blocked by a breaking behaviour in Gradle. We're usually in contact with the gradle team here or working on a fix in our build logic to resolve this.

The Elasticsearch build will fail if any deprecated Gradle API is used.

Make a change in the build

There are a few guidelines to follow that should make your life easier to make changes to the elasticsearch build. Please add a member of the `es-delivery` team as a reviewer if you're making non-trivial changes to the build.

Custom Plugin and Task implementations

Build logic that is used across multiple subprojects should considered to be moved into a Gradle plugin with according Gradle task implmentation. Elasticsearch specific build logic is located in the `build-tools-internal` subproject including integration tests.

- Gradle plugins and Tasks should be written in Java
- We use a groovy and spock for setting up Gradle integration tests. (see https://github.com/elastic/elasticsearch/blob/master/build-tools/src/testFixtures/groovy/org/elasticsearch/gradle/fixtures/AbstractGradleFuncTest.groovy)

Declaring tasks

The elasticsearch build makes use of the task avoidance API to keep the configuration time of the build low.

When declaring tasks (in build scripts or custom plugins) this means that we want to *register* a task like:

```
tasks.register('someTask') { ... }
```

instead of eagerly *creating* the task:

```
task someTask { ... }
```

The major difference between these two syntaxes is, that the configuration block of an registered task will only be executed when the task is actually created due to the build requires that task to run. The configuration block of an eagerly created tasks will be executed immediately.

By actually doing less in the gradle configuration time as only creating tasks that are requested as part of the build and by only running the configurations for those requested tasks, using the task avoidance api contributes a major part in keeping our build fast.

Registering test clusters

When using the elasticsearch test cluster plugin we want to use (similar to the task avoidance API) a Gradle API to create domain objects lazy or only if required by the build. Therefore we register test cluster by using the following syntax:

```
def someClusterProvider = testClusters.register('someCluster') { ... }
```

This registers a potential testCluster named `somecluster` and provides a provider instance, but doesn't create it yet nor configures it. This makes the gradle configuration phase more efficient by doing less.

To wire this registered cluster into a `TestClusterAware` task (e.g. `RestIntegTest`) you can resolve the actual cluster from the provider instance:

```
tasks.register('someClusterTest', RestIntegTestTask) {
    useCluster someClusterProvider
    nonInputProperties.systemProperty 'tests.leader_host', "${->
someClusterProvider.get().getAllHttpSocketURI().get(0)}"
}
```

Adding additional integration tests

Additional integration tests for a certain elasticsearch modules that are specific to certain cluster configuration can be declared in a separate so called `qa` subproject of your module.

The benefit of a dedicated project for these tests are:

- `qa` projects are dedicated two specific usecases and easier to maintain
- It keeps the specific test logic separated from the common test logic.
- You can run those tests in parallel to other projects of the build.

Using test fixtures

Sometimes we want to share test fixtures to setup the code under test across multiple projects. There are basically two ways doing so.

Ideally we would use the build-in java-test-fixtures gradle plugin. This plugin relies on having a separate sourceSet for the test fixtures code.

In the elasticsearch codebase we have test fixtures and actual tests within the same sourceSet. Therefore we introduced the `elasticsearch.internal-test-artifact` plugin to provides another build artifact of your project based on the `test` sourceSet.

This artifact can be resolved by the consumer project as shown in the example below:

```
dependencies {
  //add the test fixtures of `:providing-project` to testImplementation configuration.
  testImplementation(testArtifact(project(":fixture-providing-project')))
}
```

This test artifact mechanism makes use of the concept of component capabilities similar to how the gradle build-in `java-test-fixtures` plugin works.

`testArtifact` is a shortcut declared in the elasticsearch build. Alternatively you can declare the dependency via

```
dependencies {
  testImplementation(project(":fixture-providing-project')) {
    requireCapabilities("org.elasticsearch.gradle:fixture-providing-project-test-
artifacts")
  }
}
```

FAQ

How do I test a development version of a third party dependency?

To test an unreleased development version of a third party dependency you have several options.

How to use a maven based third party dependency via mavenlocal?

1. Clone the third party repository locally
2. Run `mvn install` to install copy into your `~/.m2/repository` folder.
3. Add this to the root build script:

```
  allprojects {
    repositories {
        mavenLocal()
    }
  }
```

4. Update the version in your dependency declaration accordingly (likely a snapshot version)

5. Run the gradle build as needed

How to use a maven built based third party dependency with jitpack repository?

https://jitpack.io is an adhoc repository that supports building maven projects transparently in the background when resolving unreleased snapshots from a github repository. This approach also works as temporally solution and is compliant with our CI builds.

1. Add the JitPack repository to the root build file:

```
   allprojects {
     repositories {
        maven { url "https://jitpack.io" }
     }
   }
```

2. Add the dependency in the following format

```
dependencies {
  implementation 'com.github.User:Repo:Tag'
}
```

As version you could also use a certain short commit hash or `master-SNAPSHOT`. In addition to snapshot builds JitPack supports building Pull Requests. Simply use PR-SNAPSHOT as the version.

3. Run the gradle build as needed. Keep in mind the initial resolution might take a bit longer as this needs to be built by JitPack in the background before we can resolve the adhoc built dependency.

NOTE

You should only use that approach locally or on a developer branch for for production dependencies as we do not want to ship unreleased libraries into our releases.

How to use a custom third party artifact?

For third party libraries that are not built with maven (e.g. ant) or provided as a plain jar artifact we can leverage a flat directory repository that resolves artifacts from a flat directory on your filesystem.

1. Put the jar artifact with the format `artifactName-version.jar` into a directory named `localRepo` (you have to create this manually)
2. Declare a flatDir repository in your root build.gradle file

```
allprojects {
  repositories {
      flatDir {
          dirs 'localRepo'
      }
  }
}
```

3. Update the dependency declaration of the artifact in question to match the custom build version. For a file named e.g. `jmxri-1.2.1.jar` the dependency definition would be `:jmxri:1.2.1` as it comes with no group information:

```
dependencies {
    implementation ':jmxri:1.2.1'
}
```

4. Run the gradle build as needed.

NOTE

As Gradle prefers to use modules whose descriptor has been created from real meta-data rather than being generated, flat directory repositories cannot be used to override artifacts with real meta-data from other repositories declared in the build. For example, if Gradle finds only `jmxri-1.2.1.jar` in a flat directory repository, but `jmxri-1.2.1.pom` in another repository that supports meta-data, it will use the second repository to provide the module. Therefore it is recommended to declare a version that is not resolveable from public repositories we use (e.g. maven central)
