Java Tips

FAQ

How do I configure SLF4J?

To configure SLF4J in Gradle project:

1. Add a log4j-test.properties under the directory of the java test.
2. Add the following snippets into your `build.gradle` file.

   ```
   test {
   systemProperty "log4j.configuration", "log4j-test.properties"
   }

   dependencies {
   shadow library.java.slf4j_api
   shadow library.java.slf4j_log4j14
   // or shadow library.java.slf4j_jdk12
   }
   ```

   Note: as of Beam 2.53.0, Beam does not support slf4j 2.x. Make sure your slf4j dependencies are of version 1.x
3. The second dependency `shadow library.java.slf4j_log4j14` is not necessary if another library already provides this dependency.
4. Check the dependency included in the dependency tree, execute:

   ```
   ./gradlew dependencies.
   ```

5. Check If you encounter an error message like the following.

   ```
   SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
   SLF4J: Defaulting to no-operation (NOP) logger implementation
   ```

   a. If so, it means there is no SLF4J.Add **`library.java.slf4j_log4j12`** or **`library.java.slf4j_jdk14`** in the `build.gradle` file.

To configure SLF4J in Maven project

1. Configure the dependency in pom.xml:

```
<properties>
<slf4j.version>1.7.30</slf4j.version>
</properties>

<dependencies>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>${slf4j.version}</version>
</dependency>

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-jdk14</artifactId>
    <version>${slf4j.version}</version>
    <!-- When loaded at runtime this will wire up slf4j to the JUL backend -->
    <scope>runtime</scope>
</dependency>
<dependencies>
```

How to format code automatically and avoid spotless errors?

1. Set up a git pre-commit hook to always autoformat code, add the following in `.git/hooks/pre-commit.`

2. Set the executable bit.
3. For more information about git hooks, go to: https://git-scm.com/docs/githooks .
4. To skip it, use `--no-verify.`
5. To disable it, use `` `chmod u-x. ``

How to run a single test?

- Example command (run from `beam` root):

```
./gradlew :examples:java:test --tests org.apache.beam.examples.subprocess.ExampleEchoPipelineTest --info
```

- To break that line down a bit:
    - `./gradlew`
        - the Gradle wrapper that runs your code. It lives in the `beam` root, so wherever you run your command from, this path needs to point there.
    - `:examples:java:test`
        - Everything before the last colon is the path from the project root to the root of the subproject the test is in (this directory will contain a `build.gradle` file)
        - The last word after the colon will always be `test` because it isn't a directory name, but the name of the Gradle task you're asking the wrapper to perform
    - `--tests`
        - this is the option that lets you declare which specific test(s) (or test suite(s)) to run, typically using their path(s) from the `src/test/java` folder of the subproject
    - `--info` (optional)
        - sets the log level to info
- For more information see the documentation below on:
    - Gradle CLI
    - Java test filtering

How to run Java Dataflow `Hello World` pipeline with compiled Dataflow Java worker.

You can dump multiple definitions for a `gcp project` name and `temp` folder. They are present since different targets use different names.

1. Before running the command, configure your gcloud credentials.
2. Add `GOOGLE_APPLICATION_CREDENTIALS` to your `env` variables.
3. Execute:

```
./gradlew :runners:google-cloud-dataflow-java:examples:preCommitLegacyWorker -
PdataflowProject=<GcpProjectName> -Pproject=<GcpProjectName> -PgcpProject=<GcpProjectName> -PgcsTempRoot=<Gcs
location in format: gs://..., no trailing slash> -PdataflowTempRoot=<Gcs location in format: gs://...>
```

```
./gradlew :runners:google-cloud-dataflow-java:examples:preCommitFnApiWorker -PdataflowProject=<GcpProjectName> -
Pproject=<GcpProjectName> -PgcpProject=<GcpProjectName>  -PgcsTempRoot=<Gcs location in format: gs://..., no
trailing slash> -PdataflowTempRoot=<Gcs location in format: gs://..., no trailing slash> -
PdockerImageRoot=<docker image store location in format gcr.io/...>
```

How to run a User Defined Pipeline - Java Direct Runner example

If you want to run your own pipeline, and in the meanwhile change beam repo code for dev/testing purposes. Here is an example for a simple runner like directRunner:

1. Put your pipeline code under the `example` folder.
2. Add the following build target to the related `build.gradle`:

   ```
   task execute(type:JavaExec) {
   main = "org.apache.beam.examples.SideInputWordCount"
   classpath = configurations."directRunnerPreCommit"
   }
   ```

There are also alternative choices, with a slight difference:

Option 1

1. Create a maven project.
2. Use the following command to publish changed code to the local repository.

   ```
    ./gradlew -Ppublishing -PnoSigning publishMavenJavaPublicationToMavenLocal
   ```

Option 2

1. Make use of Integration tests.
2. Make your user-defined pipeline part of the integration test.

How to use a snapshot Beam Java SDK version?

To use snapshot BEAM new features prior to the next Beam release, you need to;

1. Add the `apache.snapshots` repository to your `pom.xml.` Check this example .
2. Set `beam.version` to a snapshot version, e.g. "2.24.0-SNAPSHOT" or later (listed here).

Common Errors

`Continue on error`

Use the `--continue` flag makes to compileJava task and to dump all found errors, not stop on first.

IntelliJ Proto Intellisense doesn't work.

This can happen when you start IntelliJ or (in my case) after modifying `protos`.

This is not a solved problem yet. But here are some current approaches:

1. Clean `build` from console
2. Build from IntelliJ
3. Refresh Gradle Project in IntelliJ
4. Restart IntelliJ
5. Another option is if `index` is not updated with 3 or 4 steps. For more information, go to Rebuild IntelliJ project indexes.

A workaround that did the trick. Since many things were tried in the process and no clear way to reproduce the error, this might not be the correct or best step. Update steps if you find a shorter or cleaner way to do the trick.

1. Refresh `gradle` project in IntelliJ.

2. Close Intellij.
3. Clean build project from the console. Execute>

   ```
   ./gradlew clean cleanTest build -x testWebsite -x :rat -x test
   ```

4. Open IntelliJ.

Build errors due to inconsistent Gradle cache

Sometimes build fails even for the main (master) branch either using IntelliJ or command line. If it worked before but now consistently failing, most likely this is due to inconsistent Gradle cache. It could happen when switching branches back and forth. Run the build Gradle command line with "--rerun-tasks" would do the trick.

What command should I run locally before creating a pull request?

We recommend running this command, in order to catch common style issues, potential bugs (using code analysis), and Javadoc issues before creating a pull request. Running this takes 5 to 10 minutes.

```
./gradlew spotlessApply && ./gradlew -PenableCheckerFramework=true checkstyleMain checkstyleTest javadoc
spotbugsMain compileJava compileTestJava
```

If you don't run this locally Jenkins will run them during pre-submit. However, if these fail during pre-submit, you may not see the output of test failures. So doing this first is recommended to make your development process a bit smoother and iterate on your PR until it passes the pre-submit.

Dependency Upgrades

Unable to render {include} The included page could not be found.