Teamscale JaCoCo Agent

This is a Java agent that allows recording coverage, similar to JaCoCo (which is in fact used underneath). It improves on JaCoCo in the following ways:

- allow configuration via a .properties file
- allow time-interval based dumping of coverage (e.g. "dump coverage every hour") instead of only at JVM shutdown
- directly dumps XML instead of .exec files, i.e. already performs that conversion automatically
- transfers dumped XML files either to a central location for further processing or even uploads them directly to Teamscale
- writes a log file, which makes debugging problems a bit easier

Requirements

The agent might need additional memory whenever a coverage report is created. If your JVM is configured to only use very little memory, you might have to increase this limit in order for the agent to function properly. How much memory is needed depends on how much code is being analyzed and thus on how you configure the agent.

To upload coverage directly to Teamscale, a Teamscale version of 5.9.0 or higher is required.

Installing

Unzip this zip file into any folder. **All files contained in the zip file are required for the agent to work.**

Configure the agent on your application's JVM:

```
-javaagent:AGENTJAR=OPTIONS
```

Where

- `AGENTJAR` is the path to the Jar file of the Teamscale JaCoCo agent (inside the `lib` folder of this zip)
- `OPTIONS` are one or more comma-separated options for the agent in the format `key1=value1,key2=value2` and so on.

The following options are available:

General options

- `includes` (recommended): include patterns for classes. Separate multiple patterns with a semicolon. This may speed up the profiled application and reduce the size of the output XML. These patterns are matched against the Java class names. E.g. to match all classes in package `com.yourcompany` and `com.yourotherpackage` and all their subpackages you can use `*com.yourcompany.*;*com.yourotherpackage.*` (the initial star before each package name is a precaution in case your classes are nested inside e.g. a `src` folder, which might be interpreted as a part of the package name. We recommend always using this form). Make sure to include **all** relevant application code but no external libraries. For further details, please see the JaCoCo documentation in the "Agent" section.
- `excludes` (optional): exclude patterns for classes. Same syntax as the `includes` parameter. For further details, please see the JaCoCo documentation in the "Agent" section.

- `out` (optional): the path to a writable directory where the generated coverage XML files will be stored. (For details see path format section below). Defaults to the subdirectory `coverage` inside the agent's installation directory.
- `config-file` (optional): a file which contains one or more of the previously named options as `key=value` entries which are separated by line breaks. The file may also contain comments starting with `#`. (For details see path format section below)
- `logging-config` (optional): path to a logback configuration XML file (other configuration formats are not supported at the moment). Use this to change the logging behaviour of the agent. Some sample configurations are provided with the agent in the `logging` folder, e.g. to enable debug logging or log directly to the console. (For details see path format section below)
- `mode` (optional): which coverage collection mode to use. Can be either `normal` or `testwise` (Default is `normal`)

You can pass additional options directly to the original JaCoCo agent by prefixing them with `jacoco-`, e.g. `jacoco-sessionid=session1` will set the session ID of the profiling session. See the "Agent" section of the JaCoCo documentation for a list of all available options.

The `-javaagent` option MUST be specified BEFORE the `-jar` option!

Please check the produced log file for errors and warnings before using the agent in any productive setting.

The log file is written to the agent's directory in the subdirectory `logs` by default. If there is no log file at that location, it means the agent didn't even start and you have not configured it correctly.

Path format

All paths supplied to the agent can be absolute or relative to the working directory. Furthermore paths may contain ant patterns with `*`, `**` and `?`.

Options for normal mode

- `class-dir`: the path under which all class files of the profiled application are stored. Normally, this is inferred by the agent automatically. For some application, profiling performance may improve if you specify it explicitly. May be a directory or a Jar/War/Ear/... file. Separate multiple paths with a semicolon. You may also supply one or more `.txt` files with classpath entries separated by newlines (For details see path format section above)
- `interval`: the interval in minutes between dumps of the current coverage to an XML file (Default is 480, i.e. 8 hours). If set to 0 coverage is only dumped at JVM shutdown.
- `dump-on-exit`: whether a coverage report should be written on JVM shutdown (Default is true).
- `duplicates`: defines how JaCoCo handles duplicate class files. This is by default set to `WARN` to make the initial setup of the tool as easy as possible. However, this should be set to `FAIL` for productive use if possible. In special cases you can also set it to `IGNORE` to print no warnings. See the special section on `duplicates` below.
- `ignore-uncovered-classes`: Whether classes without any recorded coverage should be ignored when generating the XML coverage report. Since Teamscale assumes classes not contained in the report to have no coverage at all, this can reduce report sizes for large systems (Default is false).
- `upload-url`: an HTTP(S) URL to which to upload generated XML files. The XML files will be zipped before the upload.

- `upload-metadata`: paths to files that should also be included in uploaded zips. Separate multiple paths with a semicolon. You can use this to include useful meta data about the deployed application with the coverage, e.g. its version number.
- `teamscale-server-url`: the HTTP(S) URL of the Teamscale instance to which coverage should be uploaded.
- `teamscale-project`: the project alias or ID within Teamscale to which the coverage belongs. If not specified, the `teamscale.project` property must be specified via the `git.properties` file in at least one of the profiled JARs/WARs/EARs.
- `teamscale-user`: the username used to authenticate against Teamscale. The user account must have the "Perform External Uploads" permission on the given project.
- `teamscale-access-token`: the access token of the user.
- `teamscale-partition`: the partition within Teamscale to upload coverage to. A partition can be an arbitrary string which can be used to encode e.g. the test environment or the tester. These can be individually toggled on or off in Teamscale's UI.
- `teamscale-revision`: the source control revision (e.g. SVN revision or Git hash) that has been used to build the system under test. Teamscale uses this to map the coverage to the corresponding source code. For an alternative see `teamscale-revision-manifest-jar`.
- `teamscale-commit`: the commit (Format: `branch:timestamp`) which has been used to build the system under test. Teamscale uses this to map the coverage to the corresponding source code. Thus, this must be the exact code commit from the VCS that was deployed. For an alternative see `teamscale-commit-manifest-jar` and `teamscale-git-properties-jar`.
- `obfuscate-security-related-outputs`: boolean value determining if security critical information such as access keys are obfuscated when printing them to the console or into the log (default is true).

  If **Git** is your VCS, you can get the commit info via

```
echo `git rev-parse --abbrev-ref HEAD`:`git --no-pager log -n1 --format="%ct000"`
```

Note: Getting the branch does most likely not work when called in the build pipeline, because Jenkins, GitLab, Travis etc. checkout a specific commit by its SHA1, which leaves the repository in a detached head mode and thus returns HEAD instead of the branch. In this case the environment variable provided by the build runner should be used instead.

If **Subversion** is your VCS and your repository follows the SVN convention with `trunk`, `branches`, and `tags` directories, you can get the commit info via

```
echo `svn info --show-item url | egrep -o '/(branches|tags)/[^/]+|trunk' | egrep -o '[^/]
+$'`:`LANG=C svn info --show-item last-changed-date | date -f - +"%s%3N"`
```

- `teamscale-revision-manifest-jar` As an alternative to `teamscale-revision` the agent accepts the repository revision provided in the given jar/war's `META-INF/MANIFEST.MF` file (for details see path format section above). The revision must be supplied as an main attribute called `Revision` (preferred) or as an attribute called `Git_Commit`, which belongs to an entry called `Git`.
- `teamscale-commit-manifest-jar` As an alternative to `teamscale-commit` the agent accepts values supplied via `Branch` and `Timestamp` entries in the given jar/war's `META-INF/MANIFEST.MF` file. (For details see path format section above)
- `teamscale-git-properties-jar` As an alternative to `teamscale-commit` and/or `teamscale-project` the agent accepts values supplied via a `git.properties` file generated with the corresponding Maven or Gradle plugin and stored in a jar/war/ear/... If nothing is

configured, the agent automatically searches all loaded Jar/War/Ear/... files for a `git.properties` file. This file must contain at least the properties `git.branch` and `git.commit.time` (in the format `yyyy-MM-dd'T'HH:mm:ssZ`).
- `teamscale-message` (optional): the commit message shown within Teamscale for the coverage upload (Default is "Agent coverage upload").
- `config-file` (optional): a file which contains one or more of the previously named options as `key=value` entries which are separated by line breaks. The file may also contain comments starting with `#`. (For details see path format section above)
- `validate-ssl` (optional): defaults to true. Can be used to disable SSL validation (not recommended).
- `azure-url`: a HTTPS URL to an azure file storage. Must be in the following format: https://<account>.file.core.windows.net/<share>/(<path>). The <path> is optional; note, that in the case that the given path does not yet exists at the given share, it will be created.
- `azure-key`: the access key to the azure file storage. This key is bound to the account, not the share.
- `http-server-port`: the port at which the agent should start an HTTP server that listens for commands. The agent's REST API has the following endpoints:
  - `[GET] /partition` Returns the name of the currently configured partition name.
  - `[PUT] /partition` Sets the name of the partition name to the string delivered in the request body in plain text. This partition should be used for all followup report dumps (see `teamscale-partition`). For reports that are not directly sent to Teamscale the generated report will contain the partition name as session ID.
  - `[GET] /message` Returns the name of the currently configured commit message.
  - `[PUT] /message` Sets the commit message to the string delivered in the request body in plain text. This message should be used for all followup report dumps (see `teamscale-message`).
  - `[POST] /dump` Instructs the agent to dump the collected coverage.
  - `[POST] /reset` Instructs the agent to reset the collected coverage. This will discard all coverage collected in the current JVM session.
- `artifactory-url`: the HTTP(S) url of the artifactory server to upload the reports to. The URL may include a subpath on the artifactory server, e.g. `https://artifactory.acme.com/my-repo/my/subpath`.
- `artifactory-user` (required for artifactory): The name of an artifactory user with write access.
- `artifactory-password` (required for artifactory): The password of the user.
- `artifactory-api-key` (alternative to `artifactory-user` and `artifactory-password`) The API key for artifactory from a user with write access (c.f. Artifactory Documentation)
- `artifactory-zip-path` (optional): The path within the stored ZIP file where the reports are stored. Default is to store at root level. This can be used to encode e.g. a partition name that is parsed later on via Teamscale Artifactory connector options.
- `artifactory-git-properties-jar` (optional): Specify a Jar to search a `git.properties` file within. If not specified, Git commit information is extracted from the first found `git.properties` file. See `teamscale-git-properties-jar` for details.
- `artifactory-git-properties-commit-date-format` (optional): The Java data pattern `git.commit.time` is encoded with in `git.properties`. Defaults to `yyyy-MM-dd'T'HH:mm:ssZ`.
- `sap-nwdi-applications` needed when profiling in a SAP NetWeaver Development Infrastructure. It must be a semicolon separated list of applications. Each application is specified as

a fully qualified classname (referred to as marker class) and a Teamscale project alias or ID separated by a colon. The marker class must be guaranteed to be executed when the application is running and is unique amongst the other deployed applications. E.g. `com.company.app1.Main:app1alias;com.company.app2.Starter:ts-app2-id`. The coverage is uploaded to master at the timestamp of the last modification date of the given marker class.

Secure communication

If the connection to the Teamscale server should be established via HTTPS, a Java Trust Store can be required, which contains the certificate used by the Teamscale server for inbound HTTPS communication. Please refer to the section on HTTPS configuration in the Teamscale documentation for details on when and how to create a Java Trust Store (please note that from the viewpoint of the agent, Teamscale is the "external" server here. In order to activate usage of the trust store you need to specify the following JVM parameters

```
-Djavax.net.ssl.trustStore=<Path-to-Truststore-File>
-Djavax.net.ssl.trustStorePassword=<Password>
```

Options for testwise mode

The testwise coverage mode allows to record coverage per test, which is needed for Test Impact Analysis. This means that you can distinguish later, which test did produce which coverage. To enable this the `mode` option must be set to `testwise`.

Tests are identified by the `uniformPath`, which is a file system like path that is used to uniquely identify a test within Teamscale and should be chosen accordingly. It is furthermore used to make the set of tests hierarchically navigable within Teamscale.

In the testwise coverage mode the agent only produces an exec file that needs to be converted and augmented with more data from the test system. Please refer to the documentation for the Test Impact Analysis for more details.

There are two basic scenarios to distinguish between.

1. The system under test is restarted for every test case

To record coverage in this setting an environment variable must be set to the test's uniform path before starting the system under test. New coverage is always appended to the existing coverage file, so the test system is responsible for cleaning the output directory before starting a new test run.

- `test-env` (required): the name of an environment variable that holds the name of the test's uniform path

2. The system under test is started once

The test system (the application executing the test specification) can inform the agent of when a test started and finished via a REST API. The corresponding server listens at the specified port.

- `http-server-port` (required): the port at which the agent should start an HTTP server that listens for test events (Recommended port is 8123)
- `class-dir` (required when `tia-mode` is set to either `http` or `teamscale-upload`): the path under which all class files of the profiled application are stored. May be a directory or a Jar/War /Ear/... file. Separate multiple paths with a semicolon. (For details see path format section above)

REST API

The agent's REST API has the following endpoints:

- `[GET] /test` Returns the testPath of the current test. The result will be empty when the test already finished or was not started yet.
- `[GET] /revision` Returns the source control revision or commit the system under test was build from. This is required to upload the coverage to Teamscale at the correct point in time. The information can be supplied using any of the options `teamscale-revision`, `teamscale-commit`, `teamscale-commit-manifest-jar`, or `teamscale-git-properties-jar` above. Please note that git.properties auto-discovery is not yet supported for testwise mode.

  The response is in json format:

```
{
 "type": "COMMIT|REVISION",
 "value": "<branch>:<timestamp>|<revision>"
}
```

- `[POST] /testrun/start` If you configured a connection to Teamscale via the `teamscale-` options, this will fetch impacted tests from Teamscale and return them in the response body. The format is the same as returned by Teamscale itself. You may optionally provide a list of all available test cases in the body of the request. These will also be used to generate the testwise coverage report in `[POST] /testrun/end`. The format of the request body is:

  ```
    [
        {
          "clusterId": "<ID of the cluster the test belongs to>",
          "uniformPath": "<Unique name of the test case>",
          "sourcePath": "<Optional: Path to the source of the test>",
          "content": "<Optional: Value to detect changes to the test, e.g. hash
  code, revision, ...>"
        }
    ]
  ```

  Additionally, you may pass the following optional URL query parameters:

  - `include-non-impacted`: If this is `true`, will not perform test-selection, only test-prioritization.
  - `baseline`: UNIX timestamp in milliseconds to indicate the time since which changes should be considered. If not given, the time since the last uploaded testwise coverage report is used (i.e. the last time you ran the TIA).

- `[POST] /testrun/end` If you configured a connection to Teamscale via the `teamscale-` options and enabled `teamscale-testwise-upload`, this will upload a testwise coverage report to Teamscale.
- `[POST] /test/start/{uniformPath}` Signals to the agent that the test with the given uniformPath is about to start.
- `[POST] /test/end/{uniformPath}` Signals to the agent that the test with the given uniformPath has just finished. The body of the request may optionally contain the test execution result in json format:

```
{
  "result": "ERROR",
```

```
  "message": "<stacktrace>|<ignore reason>"
}
```

`result` can be one of:

- `PASSED` Test execution was successful.
- `IGNORED` The test is currently marked as "do not execute" (e.g. JUnit @Ignore or @Disabled).
- `SKIPPED` Caused by a failing assumption.
- `FAILURE` Caused by a failing assertion.
- `ERROR` Caused by an error during test execution (e.g. exception thrown).

(`uniformPath` and `duration` is set automatically)

The `uniformPath` parameter is a hierarchically structured identifier of the test and must be url encoded. E.g. `com/example/MyTest/testSomething` -> `http://localhost:8123/test/start/com%2Fexample%2FMyTest%2FtestSomething`.

Testwise coverage modes

You can run the testwise agent in three different modes, configured via the option `tia-mode`:

- `exec-file` (default): The agent stores the coverage in a binary `*.exec` file within the `out` directory. This is most useful when running tests in a CI/CD pipeline where the build tooling can later batch-convert all `*.exec` files and upload a testwise coverage report to Teamscale or in situations where the agent must consume as little memory and CPU as possible and thus cannot convert the execution data to a report as required by the other options. It is, however, less convenient as you have to convert the `*.exec` files yourself.
- `teamscale-upload`: the agent will buffer all testwise coverage and test execution data in-memory and upload the testwise report to Teamscale once you call the `POST /testrun/end` REST endpoint. This option is the most convenient of the different modes as the agent handles all aspects of report generation and the upload to Teamscale for you. This mode may slow down the startup of the system under test and result in a larger memory footprint than the `exec-file` mode.
- `http`: the agent converts the coverage collected during a test in-process and returns it as a JSON in the response to the `[POST] /test/end/...` request. This allows the caller to handle merging coverage of multiple tests into one testwise coverage report, e.g. in situations where more than one agent is running at the same time (e.g. profiling across multiple microservices.) This option may slow down the startup of the system under test and result in a larger memory footprint than the `exec-file` mode.

  The response format looks like this:

  ```
  {
    "uniformPath": "com/example/MyTest/testSomething",
    "duration": 0.025,
    "result": "PASSED",
    "paths": [
      {
        "path": "com/example/project",
        "files": [
          {
            "fileName": "Calculator.java",
            "coveredLines": "20-24,26,27,29"
          },
          {
  ```

```
          "fileName": "SomeOtherClass.java",
          "coveredLines": "26-28"
        }
      ]
    }
  ]
}
```

(`duration` and `result` are included if you provided a test execution result in the request body)

Additional steps for WebSphere

Register the agent in WebSphere's `startServer.bat` or `startServer.sh`. Please also apply this additional JVM parameter:

```
-Xshareclasses:none
```

This option disables a WebSphere internal class cache that causes problems with the profiler.

Please set the agent's `includes` parameter so that the WebSphere code is not being profiled. This ensures that the performance of your application does not degrade.

Also consider to use the `config-file` option as WebSphere 17 and probably other versions silently strip any option parameters exceeding 500 characters without any error message, which can lead to very subtle bugs when running the profiler.

Additional steps for JBoss

Register the agent in the `JAVA_OPTS` environment variable in the `run.conf` file inside the JBoss installation directory.

Please set the agent's `includes` parameter so that the JBoss code is not being profiled. This ensures that the performance of your application does not degrade.

Additional steps for Wildfly

Register the agent in the `JAVA_OPTS` environment variable in the `standalone.conf` or `domain.conf` file inside the Wildfly installation directory - depending on which "mode" is used; probably standalone.

Please set the agent's `includes` parameter so that the Wildfly code is not being profiled. This ensures that the performance of your application does not degrade.

Additional steps for Tomcat/TomEE

Register the agent in the `CATALINA_OPTS` environment variable inside the `bin/setenv.sh` or `bin/setenv.bat` script in the Tomcat installation directory. Create this file if it does not yet exist.

Please set the agent's `includes` parameter so that the Tomcat code is not being profiled. This ensures that the performance of your application does not degrade.

To ensure the Tomcat process is properly shutdown you need to add `CATALINA_PID="$CATALINA_BASE/bin/catalina.pid"` to the `bin/setenv.sh` script and stop Tomcat via `catalina stop -force`. This makes Tomcat store the process ID of the server in the specified file and uses it to kill the process when stopped.

Additional steps for Glassfish

You have two options to register the JVM option:

1. Register the agent via the `asadmin` tool.
2. Register the agent by manually editing `domain.xml` and adding a `jvm-options` element.

When using the `asadmin` tool, some characters need to be escaped with a backslash, see this StackOverflow answer.

Afterwards, restart Glassfish. Please verify the setup with `asadmin`'s `list-jvm-options` command.

Additional steps for Jetty

Register the agent by setting a `JAVA_OPTIONS` variable such that the Jetty process can see it.

Additional steps for SAP NetWeaver Java

Please note that the Teamscale JaCoCo Agent requires at least Java 8, which is part of NetWeaver Java 7.50. Prior versions of NetWeaver Java are not supported by the Teamscale JaCoCo Agent.

In order to set the JVM agent parameter, you need to use the NetWeaver Administrator to navigate to `Configuration` - `Infrastructure` - `Java System Properties` and add an additional JVM parameter. Use the search field to search for `agent`, and select the `-javaagent:<jarpath>[=<options>]` entry. In the `Name` field, enter the first part, until (including) the `.jar`, and provide all the options (*without* the leading `=`) in the `Value` field. We advise to only set the `config-file` option here, and provide all other options via the config file. Choose `Add` and then `Save`, and restart the Java server from the SAP Management Console.

Additional steps for Java Web Start

Please ask CQSE for special tooling that is available to instrument Java Web Start processes.

Store Commit in Jar file

If you are using Git, you can use either a Maven or Gradle plugin to store the commit in any Jar/War/Ear/... file and tell the agent to read it via `teamscale-git-properties-jar`.

Alternatively, it is also convenient to use the MANIFEST entries via `teamscale-commit-manifest-jar` to link artifacts to commits, especially when tests are executed independently from the build. The following assumes that we are using a Git repository.

Maven

To configure this for the maven build add the following to your top level `pom.xml`.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId> <!-- Works also for the maven-jar-plugin -->
    ...
    <configuration>
         ...
        <resourceEncoding>UTF-8</resourceEncoding>
        <archive>
            <manifest>
                <addDefaultImplementationEntries>true</addDefaultImplementationEntries>
                <addDefaultSpecificationEntries>true</addDefaultSpecificationEntries>
            </manifest>
            <manifestEntries>
                <Branch>${branch}</Branch>
                <Timestamp>${timestamp}</Timestamp>
            </manifestEntries>
        </archive>
    </configuration>
</plugin>
```

When executing maven pass in the branch and timestamp to Maven:

```
mvn ... -Dbranch=$BRANCH_NAME -Dtimestamp=$(git --no-pager log -n1 --format="%ct000")
```

Gradle

```
plugins {
        id 'org.ajoberstar.grgit' version '2.3.0'
}

jar {
        manifest {
                attributes 'Branch': System.getProperty("branch")
                attributes 'Timestamp': grgit.log {
                        maxCommits = 1
                }.first().dateTime.toInstant().toEpochMilli()
        }
}
```

```
./gradlew jar -Dbranch=master
```

Multi-project upload

It is possible to upload the same coverage file to multiple Teamscale projects for different commits. In this case, the `teamscale.project` property has to be provided in each of the profiled Jar/War/Ear/... files via the contained `git.properties` file. For example, the `git.properties` file can be generated using the gradle-git-properties Gradle plugin:

```
gitProperties {
    customProperty 'teamscale.project', 'my-awesome-project'
}
```

The same can be achieved for Maven using, e.g., the git-commit-id Maven plugin:

```
 <resources>
        <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
        </resource>
</resources>
```

You need to include a properties file with unresolved property values for ${teamscale.project} and ${git.commit.id} and provide the corresponding values via the `pom.xml` file(s) of your artifact(s). Example:

```
git.commit.id=${git.commit.id}
teamscale.project=${teamscale.project}
```

Please note that the commit must be provided via the `git.properties` in each of the profiled Jar/War/Ear... files when specifying the `teamscale.project` this way. In case `teamscale-project` is provided via the agent properties, its value takes precedence over any `teamscale.project` value provided via the `git.properties`.

`duplicates`

The underlying JaCoCo coverage instrumentation tooling relies on fully qualified class names to uniquely identify classes. However, in practice, applications are often deployed with multiple versions of the same class. This can happen, e.g. if you use the same library in different versions in sub-projects of your code.

At runtime, it is not deterministic, which of these versions will be loaded by the class loader. Thus, when trying to convert the recorded coverage back to covered source code lines, JaCoCo cannot determine which of the two versions was actually profiled.

By default, the agent is configured to only log a warning in these cases and simply pick one of the versions at random. Thus, the reported coverage for such files may not be accurate and may even be totally unusable. It is thus desirable to fix these warnings by ensuring that only one version of each class is deployed with your application. This has to be fixed in your build process.

Please refer to this StackOverflow post and the JaCoCo FAQ for more information.

Docker

If you'd like to set up the agent for Docker, please refer to our guide on Java with Docker profiling

One-time conversion of .exec files to .xml

This tool is also useful in case you are using the plain JaCoCo agent.

Converting `.exec` files produced by raw JaCoCo to XML can be cumbersome. You can run the `bin/convert` tool to perform this conversion for you. Run `bin/convert --help` to see all available command line options.

This is especially useful since this conversion does allow for duplicate class files by default, which the raw JaCoCo conversion will not allow.

The caveats listed in the above `ignore-duplicates` section still apply!

Troubleshooting

My application fails to start after registering the agent

Most likely, you provided invalid parameters to the agent. Please check the agent's directory for a log file. If that does not exist, please check stdout of your application. If the agent can't write its log file, it will report the errors on stdout.

Produced coverage files are huge

You're probably profiling and analyzing more code than necessary (e.g. third-party libraries etc). Make sure to set restrictive include/exclude patterns via the agent's options (see above).

Enable debug logging to see what is being filtered out and fine-tune these patterns.

I do not have access to the class files

In that case simply *don't* specify a `class-dir` option.

Error: "Can't add different class with same name"

This is a restriction of JaCoCo. See the above section about `ignore-duplicates`. To fix this error, it is best to resolve the underlying problem (two classes with the same fully qualified name but different code). If this is not possible or not desirable, you can set `ignore-duplicates=true` in the agent options to turn this error into a warning. Be advised that coverage for all classes that produce this error/warning in the log may have inaccurate coverage values reported.

Please refer to this StackOverflow post and the JaCoCo FAQ for more information.

How to change the log level

Set an appropriate logback logging configuration XML in the agent options:

```
logging-config=/PATH/TO/logback-config.xml
```

You can find example logging configurations in the logging folder in the agent installation directory.

How to enable debug logging

An appropriate logging configuration is shipped with the agent under `logging/logback.debug.xml`. Set the agent option

```
logging-config=/PATH/TO/AGENT/INSTALL/DIRECTORY/logging/logback.debug.xml
```

The debug logs are written to `/PATH/TO/AGENT/INSTALL/DIRECTORY/logs`.

How to see which files/folders are filtered due to the `includes` and `excludes` parameters

Enable debug logging in the logging config. Warning: this may create a lot of log entries!

Error: "The application was shut down before a commit could be found", despite including a git.properties file in your jar/war/...

When using application servers, the `git.properties` file in your jar/war/... might not be detected automatically, which results in an "The application was shut down before a commit could be found" error. To resolve the problem, try specifying `teamscale-git-properties-jar` explicitly.