Integration Testing Framework Background Guide

Table of Contents

1. Overview of the Current Situation

We will use some terms in this document:

- The word **cache** will be used for the local repository (`$HOME/.m2/repository`).

1.1. Maven Invoker Plugin

In Maven Invoker Plugin the following issues exist:

- Parallelizing does not work and is not easy to integrate based on current concept and code base.
  - Apart from being implemented it would hard to express the prevention of parallel execution

in some situations.

- Separating caches for each build is hard to implement
- Get a common cache for a set of integration tests is even harder.
- A Concept like `BeforeEach` or `BeforeAll` is current not really possible.
  - The concept with `setup` project is not correctly working at the moment.
- Writing integration tests forces one to write in Groovy or Beanshell.
  - This means to enhance the number of dependencies. In days of Java 5 until 7 it had been an advantage to use Groovy with it's supports for closures etc. which made it simpler and easier to write things for integration tests, but since JDK 8 it is not necessary anymore.
- Integrationtest are not that expressive as they should be.
- Violation of separation of concern paradigm
  - Conditions
    - Assertions are hard to express cause one implicit assertion is that a build has to be successful (can be changed if necessary)
    - Conditions for the execution of a test for example are:
      - should be executed only on JDK11
      - should be executed only on Maven 3.3.9 and above
      - Several other conditions
    - are expressed within a single file `invoker.properties`.

1.2. Maven Verifier Plugin

The Maven Verifier Plugin is intended to write tests to check for the existence of files or the absence of files but in the end it is very limited.

The Maven Verifier is intended to write integration tests for Maven ...

1.3. Maven Verifier Component

The Maven Verifier is intended to write a kind of tests:

- You can set the command line parameters for an executed instance of Maven like `-s`, `-X` etc.
- Execute goals like `package` or alike.
- It contains some methods like `assertFilePresent`, `assertFileMatches`, `verifyArtifactPresence` etc. but not a comprehensive set of methods.
- Some parts are like Maven Invoker Plugin for example starting an external process with Maven (something like starting Maven on command line.).
- Is JUnit 3 based.
- Manually maintained TestSuite to execute all integration tests of Maven Core.
- Each Testcase must be expressed by a separate Test class.

- Manually implemented conditionally execution.
- Conditions for execution only based on a self implemented constructor part which defines the Maven version under which it should run or not.

1.4. Maven Plugin Testing Harness

The Maven Plugin Testing Harnessis intended to write tests for using parameters correctly and several other setup situations but the test setup is separated into a unit test like part in code and a part which is pom like

- It's bound to versions of Maven core which might caused issues during testing with other versions of Maven.
- https://maven.apache.org/plugin-testing/maven-plugin-testing-harness/getting-started/index.html
- Also JUnit 4 based.

1.5. Mock Repository Manager

Currently it's only possible to have a single instance of the mock repository manager running which is based on the limited concept cause we need to define it in the `pom.xml`. Of course we could start two or more instances but this would make the setup more or less unreadable.

1.6. Why not Spock?

So you might ask why not using Spock or any other testing framework for such purposes? Let me summarize the different aspects I had in mind:

- People often tend to write Java code (which is valid), cause they don't know Groovy or don't want to learn a new language just to write tests. This means in the end: Why Groovy? Just use Java.
- It's much easier for new contributors/devs to get into the project if you only need to know Java to write plugins, unit tests and integration tests. So removing a supplemental barrier will help.
- Support for most recent Java versions which is a complete blocker for the Apache Maven project, cause the Apache Maven Project is running builds in a very early stage (Early access) which would block us (see our builds for example Apache Maven EAR Plugin Builds). Currently spock is not yet tested/build against JDK11+ ? So having a Testing framework which might not work on most recent versions is a complete blocker.
- In earlier days I would have argued to use Spock based on the language features but since JDK8 I don't see any advantage in using Groovy over Java anymore.
- Spock does not support parallelizing of tests (full blocker for me)
- Good IDE Support for Groovy is at the moment only given in IDEA IntelliJ as well as for DSL support for Spock. That would block many people. This blocker based on the usage of a particular IDE is not acceptable for an open source project like the Apache Maven Project and from my point of view as an Apache Maven PMC member this is simply a no go.

1.7. Conclusion

It is needed to have a combination of Maven Invoker Plugin, Maven Verifier etc. into a single Testing framework which should make it possible to make integration tests easier to write and make them more expressive about what the intention of what a test exactly is.

It looks like a good solution to use existing frameworks like JUnit Jupiter and assertions like AssertJ library to express what it's needed. This in result will give automatically many advantages for example the integration into the IDE as well as writing the tests in Java code and furthermore opens easy ways to use existing Java libraries.

Using JUnit Jupiter as base will solve lot of things which are already supported by JUnit Jupiter like conditional execution of Tests based on JRE or possible deactivation based on properties etc.

Based on AssertJ it could be easy to express the assertions for test results in many ways and can also being enhanced by writing custom assertions.

2. Basic Idea

The expressiveness of tests is a very important part of writing integration tests or test in general. If a test is not easy to understand it is very likely not being written.

Let us take a look into the following code snippet which is an idea how an integration test for a Maven Plugins/Maven Extensions/Maven-Core could look like:

```
import static org.assertj.core.api.Assertions.assertThat;

import org.apache.maven.jupiter.extension.MavenJupiterExtension;
import org.apache.maven.jupiter.extension.MavenTest;
import org.apache.maven.jupiter.extension.maven.MavenProjectResult;

@MavenJupiterExtension
class FirstMavenIT {

  @MavenTest
  void the_first_test_case(MavenProjectResult result) {
    assertThat(result)
      .build()
        .isSuccessful()
      .and()
      .project()
        .hasTarget()
          .withEarFile()
            .containsOnlyOnce("META-INF/MANIFEST.MF")
        .log()
          .info().contains("Writing data to file")
      .cache()
          .withEarFile("G:A:V")
          .withPomFile("G:A:V")
          .withMetadata().contains("xxx");
  }
}
```

3. Concept

The idea was to create an JUnit Jupiter extension which will support writing of integration tests for Maven plugins etc. in a convenient way. Furthermore writing custom assertions with AssertJ library makes it easier to express the intention of a test.

Basic Idea is currently similar to maven-invoker-plugin: Another option would be to combine this with Docker containers which run Maven. Extension starts the appropriate Maven version via ProcessBuilder with parameters in it's own directory (`target/maven-it/ …`)

- Separate
- Existing repository which contains already installed artifacts for special cases (see Versions Maven Plugin a lot of test cases need special artifacts in repository for integration tests). Using a directory default: `local-repo`. ?

4. Example

The following integration test is a basic skeleton of an integration test which implies some conventions which will be describe within the following paragraphs.

FirstIT.java

```
package org.it;
import org.apache.maven.jupiter.extension.MavenJupiterExtension;
import org.apache.maven.jupiter.extension.MavenTest;
import org.apache.maven.jupiter.extension.maven.MavenExecutionResult;

@MavenJupiterExtension
class FirstIT {

  @MavenTest
  void first(MavenExecutionResult result) {
  }

  @MavenTest
  void second(MavenExecutionResult result) {
  }
}
```

The directory structure of an integration test will look like this. This is by convention the same as for any kind of unit- or integration-test in Maven or more in general in Java projects.

```
src
 +-- test
      +-- java
            +-- org
                 +-- it
                      +-- FirstIT.java
```

The convention is simply by mapping the method name (including the package name) into a directory. The `resources` directory is the location where to find the project for the integration tests. Basic start is the class name `FirstIT` which defines the base directory for all test cases.

In Ma The intermediate directory `maven-its` is intended to separate the usual resources from the integration test resources.

```
src
 +-- test
      +-- resources
            +-- maven-its
                +-- org
                     +-- it
                          +-- FirstIT
```

Now we have the need to separate each test case from each other which is done via the method name of the test case within the test class `FirstIT` which has the methods `first` and `second` in our examples. This will look like the following:

```
src
 +-- test
      +-- resources
            +-- maven-its
                +-- org
                     +-- it
                          +-- FirstIT
                                +- first
                                     +- src
                                     +- pom.xml
                                +- second
                                     +- src
                                     +- pom.xml
```

During the execution of the integration tests the following directories will be created within the `target` directory:

```
target
 +- maven-its
        +- org
           +- it
              +- FirstIT
                  +- first
                        +- .m2/
                        +- project
                        +- mvn-stdout.log
                        +- mvn-stderr.log
                        +- other logs
                  +- second
                        +- .m2/
                        +- project
                        +- mvn-stdout.log
                        +- mvn-stderr.log
                        +- other logs
```

Based on the above you can see that each test case (method within the class) has it's own local cache (`.m2/repository`). You see the resulting project is built within the `project` folder to make separated from log files and local cache. The result of this setup is that each test case is completely separated from each other test case and gives us an easy way to parallelize the integration test cases in a simple way.

It is possible to define the cache for several test cases globally which can simply being done by using the following annotation `@MavenRepository`. This give the opportunity to make different tests share the same cache which is like a usual setup for a user on a local machine which can be used to test different scenarios. The default behaviour is that each test case has it's own local cache `.m2/repository`.

One very important thing is to mention that if you define `@MavenRepository` as given in the following

example you have to be aware of that those test cases running by default in parallel which mean you have to limit the thread usage via `@Execution(ExecutionMode.SAME_THREAD)` otherwise it could happen you might get strange errors.

FirstMavenIT.java

```
package org.it;

import org.apache.maven.jupiter.extension.MavenJupiterExtension;
import org.apache.maven.jupiter.extension.MavenRepository;
import org.apache.maven.jupiter.extension.MavenTest;
import org.apache.maven.jupiter.extension.maven.MavenExecutionResult;
import org.junit.jupiter.api.parallel.Execution;
import org.junit.jupiter.api.parallel.ExecutionMode;

@MavenJupiterExtension
@MavenRepository
@Execution(ExecutionMode.SAME_THREAD)
class FirstIT {

  @MavenTest
  void first(MavenExecutionResult result) {
  }

  @MavenTest
  void second(MavenExecutionResult result) {
  }
}
```

Sometimes it could be useful to setup a number of project together to test things related to usage of other other artifacts or other projects etc. this can be achieved by using the following setup:

MavenIntegrationIT.java

```
package org.it;

import static org.apache.maven.jupiter.assertj.MavenITAssertions.assertThat;

import org.apache.maven.jupiter.extension.MavenJupiterExtension;
import org.apache.maven.jupiter.extension.MavenRepository;
import org.apache.maven.jupiter.extension.MavenTest;
import org.apache.maven.jupiter.extension.maven.MavenExecutionResult;
import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.TestMethodOrder;

@MavenJupiterExtension
@MavenRepository
@TestMethodOrder(OrderAnnotation.class)
class MavenIntegrationIT {

  @MavenTest(goals = {"install"})
  @Order(10)
  void setup(MavenExecutionResult result) {
    assertThat(result).isSuccessful();
  }

  @MavenTest(goals = {"install"})
  @Order(20)
  void setup_2(MavenExecutionResult result) {
    assertThat(result).isSuccessful();
  }

  @MavenTest
  void first_integration_test(MavenExecutionResult result) {
    assertThat(result).isSuccessful();
  }
}
```

Based on the given annotation `@MavenRepository` this will define a global cache for all test cases within the given test class `MavenIntegrationIT`.

So based on the above test case you will get a resulting directory structure which looks like this:

```
target
 +- maven-its
       +- org
          +- it
             +- MavenIntegrationIT
                 +- .m2/
                 +- setup
                       +- project
                       +- mvn-stdout.log
                       +- mvn-stderr.log
                       +- other logs
                 +- setup_2
                       +- project
                       +- mvn-stdout.log
                       +- mvn-stderr.log
                       +- other logs
                 +- first_integration_test
                       +- project
                       +- mvn-stdout.log
                       +- mvn-stderr.log
                       +- other logs
```

There are two things to mention. First the cache which is common for all given tests cases `setup`, `setup_2` and for `first_integration_test`. Furthermore the definition of the order of execution given by using `@Order(10)` which defines the order of execution for those test cases which are used as setup projects for the real test case `first_integration_test`. This makes it easy possible define any kind of setup projects for a bigger complexer test case.

Separate repository which contains already installed artifacts `local-repo`:

Think how to make the build use it?

```
src
 +-- test
      +-- resources
            +-- maven-its
                +-- org
                     +-- it
                          +-- FirstIT
                                +- .local-repo
                                +- first
                                     +- src
                                     +- pom.xml
                                +- second
                                     +- src
                                     +- pom.xml
```

5. Execution of Maven itself

- How to get the Maven version which is defined?
  - Define within the same pom file you run your tests?
  - Ok could be downloaded from Central?
  - how to handle repository managers?
- Where to get configured all the avialble Maven versions? On the system? or should we simply download it always to be sure?

6. Parameter Injection

Possible options:

- Information about the built project
  - version, GAV etc. maybe the whole POM tree ?
  - think more in details?
- Logging output
  - Stdout
  - StdErr
  - Log Output as Stream or after finished running
  - Convenience methods to get information from the log
    - `isInfo()` which relates to `[INFO] ..` Think about this?
    - Some things to get output from plugins etc.???
- Access to the cache directory
  - With convenience methods to access artifacts/content of artifacts
  - ???
- general build result.

6.1. MavenExecutionResult

- MavenExecutionResult
  - isSuccessful() `BUILD SUCCESS`
  - isError() `[ERROR]⋯.`
  - is

ThirdMavenIT.java

```
@MavenJupiterExtension
class FirstMavenIT {

  @MavenTest
  void first_test_case(MavenExecutionResult execResult) {
    assertThat(execResult).isSuccessful();
  }
  @MavenTest
  void second_test_case(MavenExecutionResult execResult) {
    assertThat(execResult).isFailed();
  }

}
```

6.2. Run Conditionally Integration Tests

You might want to run an integration test only for a particular Maven version for example running only for Maven 3.6.0?

ForthMavenIT.java

```
import static org.apache.maven.jupiter.assertj.MavenExecutionResultAssert.assertThat;
import static org.apache.maven.jupiter.extension.maven.MavenVersion.M3_0_5;
import static org.apache.maven.jupiter.extension.maven.MavenVersion.M3_6_0;

import org.apache.maven.jupiter.extension.DisabledForMavenVersion;
import org.apache.maven.jupiter.extension.EnabledForMavenVersion;
import org.apache.maven.jupiter.extension.MavenJupiterExtension;
import org.apache.maven.jupiter.extension.MavenTest;
import org.apache.maven.jupiter.extension.maven.MavenExecutionResult;

@MavenJupiterExtension
class FirstMavenIT {

  @MavenTest
  @EnabledForMavenVersion(M3_6_0)
  void first_test_case(MavenExecutionResult execResult) {
    assertThat(execResult).isSuccessful();
  }

  @DisabledForMavenVersion(M3_0_5)
  @MavenTest
  void second_test_case(MavenExecutionResult execResult) {
    assertThat(execResult).isFailure();
  }

}
```

So not run some tests on particular Java version can be handled via usual JUnit Jupiter things like:

FivthMavenIT.java

```
import static org.apache.maven.jupiter.assertj.MavenITAssertions.assertThat;
import static org.apache.maven.jupiter.extension.maven.MavenVersion.M3_0_5;
import static org.apache.maven.jupiter.extension.maven.MavenVersion.M3_6_0;

import org.apache.maven.jupiter.extension.DisabledForMavenVersion;
import org.apache.maven.jupiter.extension.EnabledForMavenVersion;
import org.apache.maven.jupiter.extension.MavenJupiterExtension;
import org.apache.maven.jupiter.extension.MavenTest;
import org.apache.maven.jupiter.extension.maven.MavenExecutionResult;
import org.junit.jupiter.api.condition.DisabledOnJre;
import org.junit.jupiter.api.condition.JRE;

@MavenJupiterExtension
@DisabledOnJre(JRE.JAVA_10)
class FirstMavenIT {

  @MavenTest
  @EnabledForMavenVersion(M3_6_0)
  void first_test_case(MavenExecutionResult execResult) {
    assertThat(execResult).isSuccessful();
  }

  @DisabledForMavenVersion(M3_0_5)
  @MavenTest
  void second_test_case(MavenExecutionResult execResult) {
    assertThat(execResult).isFailure();
  }
}
```

7. Assertions in Maven Tests

What kind of assertions do we need to express:

- Build itself has successfully ended or failed. (Return code? enough?)
- Log File contains several information
  - Different levels `INFO`, `WARN` or `ERROR`..
    - contains simply one or more lines text
    - contains only once or multiple appearance of texts
- StdErr output contains particular output or should not contain particular output.
- The `target` directory of the built project contains either:
  - particular files
    - simply exist/do not exist?

        - should exist or should not exist
        - The files contain particular content? for example or in general directory within the file `MANIFEST.MF`.
        - A packaged file special content?
    - directories
    - ??

SixthMavenIT.java

```
import static org.apache.maven.jupiter.assertj.MavenExecutionResultAssert.assertThat;

import org.apache.maven.jupiter.extension.MavenJupiterExtension;
import org.apache.maven.jupiter.extension.MavenTest;
import org.apache.maven.jupiter.extension.maven.MavenExecutionResult;

@MavenJupiterExtension
class FirstMavenIT {

  @MavenTest
  void first_test_case(MavenExecutionResult execResult) {
    assertThat(execResult).isSuccessful();
  }

  @MavenTest
  void second_test_case(MavenExecutionResult result) {
    assertThat(result).isFailed().log().contains().plugin("G:A:V");
    assertThat(result)
      .isSuccessful()
      .and()
      .project("G:A:V")
        .module("G:A:V")
          hasTarget().withJarFile().metainf
  }
}
```

7.1. Target Directory Handling

SeventhMavenIT.java

```
import static org.apache.maven.jupiter.assertj.MavenProjectResultAssert.assertThat;

import org.apache.maven.jupiter.extension.MavenJupiterExtension;
import org.apache.maven.jupiter.extension.MavenTest;
import org.apache.maven.jupiter.extension.maven.MavenProjectResult;

@MavenJupiterExtension
class FirstMavenIT {

  @MavenTest
  void second_test_case(MavenProjectResult project) {
    assertThat(project).hasTarget()
        .withEarFile()
        .containsOnlyOnce(
            "META-INF/application.xml",
            "META-INF/appserver-application.xml"
        );
  }

  @MavenTest
  void third_test_case(MavenProjectResult project) {
    assertThat(project).hasTarget()
        .withEarFile()
        .doesNotContain("commons-io-1.4.jar")
        .containsOnlyOnce(
            "commons-lang-commons-lang-2.5.jar",
            "META-INF/application.xml",
            "META-INF/MANIFEST.MF"
        );
  }
}
```

7.2. New Ideas

The basic idea is to have the assertions based on an entry point which is `MavenExecutionResultAssert` related to `MavenExecutionResult`.

The following are example how an integration test could look like:

UnknownMavenIT.java

```
import static org.apache.maven.jupiter.assertj.MavenProjectResultAssert.assertThat;

import org.apache.maven.jupiter.extension.MavenJupiterExtension;
import org.apache.maven.jupiter.extension.MavenTest;
import org.apache.maven.jupiter.extension.maven.MavenProjectResult;

@MavenJupiterExtension
class FirstMavenIT {

  @MavenTest
  void third_test_case(MavenProjectResult project) {
    assertThat(project)
      .hasCache()
        .withEarFile("G:A:V").containsOnlyOnce("...")
        .withJarFile("...").contains("..")
        .withPomFile("g:a:v:c").containsDependency("xxx")
        .withArchive(".tar.gz").contains("...");
    assertThat(project).log().contains("...")
    assertThat(project).hasModule("A:G").hasTarget().withEarFile()....
    assertThat(project).build().isSuccessful().hasTarget()
  }
}
```

7.3. Things which do not work yet (not complete!)

Later we will create an plugin for the purpose an can inject the information into the test cases as we already did like in Maven Invoker Plugin.

This is:

- Currently it is not possible to define the version Maven only within the test case. Unfortunately we have to define it in the Maven pom which is used to download the needed package from Central.

8. TODO List

8.1. Support for running with several Maven Versions

- Currently we are limited to run under the Maven version which is used by running the integration tests.
- We need to consider where we ran tests with different versions of Maven to check compatibility for things. Something like this:
- Based on the above requirements the following question will arise:
  - Where to download the appropriate Apache Maven versions?

    - Handle each test case separately into a separate directory to keep them independent.

MultiVersionIT.java

```
import static org.assertj.core.api.Assertions.assertThat;

import org.apache.maven.jupiter.extension.MavenJupiterExtension;
import org.apache.maven.jupiter.extension.MavenTest;
import org.apache.maven.jupiter.extension.maven.MavenProjectResult;

@MavenJupiterExtension
@MavenVersion({3_0_5, 3_3_9})
class FirstMavenIT {

  @MavenTest
  void third_test_case(MavenProjectResult project) {
    assertThat(project)
      ...
  }
}
```

- Defining a range for Maven versions which will be used to execute the tests.

MultiVersionIT.java

```
import static org.assertj.core.api.Assertions.assertThat;

import org.apache.maven.jupiter.extension.MavenJupiterExtension;
import org.apache.maven.jupiter.extension.MavenTest;
import org.apache.maven.jupiter.extension.maven.MavenProjectResult;

@MavenJupiterExtension
@MavenVersionRange(from = 3_0_5, upto=3_6_3)
class FirstMavenIT {

  @MavenTest
  void third_test_case(MavenProjectResult project) {
    assertThat(project)
      ...
  }
}
```

9. Comparison

- Testing parallelization looks already very good. The following run is using parallel execution of the tests:

```
[INFO]
[INFO] --- maven-failsafe-plugin:2.22.1:integration-test (default) @ maven-ear-plugin
---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running org.apache.maven.plugins.ear.it.EARIT
[WARNING] Tests run: 15, Failures: 0, Errors: 0, Skipped: 2, Time elapsed: 21.297 s -
in org.apache.maven.plugins.ear.it.EARIT
[INFO]
[INFO] Results:
[INFO]
[WARNING] Tests run: 15, Failures: 0, Errors: 0, Skipped: 2
[INFO]
[INFO]
[INFO] --- maven-checkstyle-plugin:3.0.0:check (checkstyle-check) @ maven-ear-plugin
---
[INFO] There are 3 errors reported by Checkstyle 6.18 with config/maven_checks.xml
ruleset.
[INFO] Ignored 3 errors, 0 violation remaining.
[INFO]
[INFO] --- maven-failsafe-plugin:2.22.1:verify (default) @ maven-ear-plugin ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  32.282 s
[INFO] Finished at: 2019-12-03T18:51:21+01:00
[INFO] ------------------------------------------------------------------------
```

- The usual way via maven-invoker

```
[INFO]
[INFO] --- maven-invoker-plugin:3.2.1:integration-test (integration-test) @ maven-ear-
plugin ---
[INFO] Building: skinny-wars-filenamemapping-full/pom.xml
[INFO] run post-build script verify.bsh
[INFO]           skinny-wars-filenamemapping-full/pom.xml ......... SUCCESS (4.1 s)
[INFO] Building: jboss/pom.xml
[INFO] run post-build script verify.bsh
[INFO]           jboss/pom.xml ..................................... SUCCESS (1.6 s)
[INFO] Building: skinny-wars/pom.xml
[INFO] run post-build script verify.bsh
[INFO]           skinny-wars/pom.xml ............................... SUCCESS (2.3 s)
[INFO] Building: transitive-excludes/pom.xml
[INFO] run post-build script verify.bsh
[INFO]           transitive-excludes/pom.xml ....................... SUCCESS (1.6 s)
[INFO] Building: MEAR-198/pom.xml
[INFO] run post-build script verify.bsh
```

```
[INFO]           MEAR-198/pom.xml ............................... SUCCESS (1.7 s)
[INFO] Building: non-skinny-wars/pom.xml
[INFO] run post-build script verify.bsh
[INFO]           non-skinny-wars/pom.xml ........................ SUCCESS (2.3 s)
[INFO] Building: filenamemapping-usage-fail/pom.xml
[INFO] run post-build script verify.groovy
[INFO]           filenamemapping-usage-fail/pom.xml .............. SUCCESS (2.5 s)
[INFO] Building: MEAR-243-skinny-wars-provided/pom.xml
[INFO] run post-build script verify.bsh
[INFO]           MEAR-243-skinny-wars-provided/pom.xml ........... SUCCESS (2.3 s)
[INFO] Building: basic/pom.xml
[INFO] run post-build script verify.bsh
[INFO]           basic/pom.xml ................................... SUCCESS (1.7 s)
[INFO] Building: packaging-includes/pom.xml
[INFO] run post-build script verify.bsh
[INFO]           packaging-includes/pom.xml ...................... SUCCESS (1.7 s)
[INFO] Building: resource-custom-directory/pom.xml
[INFO] run post-build script verify.bsh
[INFO]           resource-custom-directory/pom.xml ............... SUCCESS (1.6 s)
[INFO] Building: skinny-wars-javaee5/pom.xml
[INFO] run post-build script verify.bsh
[INFO]           skinny-wars-javaee5/pom.xml ..................... SUCCESS (2.9 s)
[INFO] Building: skinny-wars-filenamemapping-no-version/pom.xml
[INFO] run post-build script verify.bsh
[INFO]           skinny-wars-filenamemapping-no-version/pom.xml ... SUCCESS (2.3 s)
[INFO] Building: same-artifactId/pom.xml
[INFO] run post-build script verify.groovy
[INFO]           same-artifactId/pom.xml ......................... SUCCESS (3.4 s)
[INFO] Building: packaging-excludes/pom.xml
[INFO] run post-build script verify.bsh
[INFO]           packaging-excludes/pom.xml ...................... SUCCESS (1.7 s)
[INFO] Building: descriptor-encoding/pom.xml
[INFO] run post-build script verify.groovy
[INFO]           descriptor-encoding/pom.xml ..................... SUCCESS (2.0 s)
[INFO]
[INFO] --- maven-failsafe-plugin:2.22.1:integration-test (default) @ maven-ear-plugin
---
[INFO] Tests are skipped.
[INFO]
[INFO] --- maven-checkstyle-plugin:3.0.0:check (checkstyle-check) @ maven-ear-plugin
---
[INFO] There are 3 errors reported by Checkstyle 6.18 with config/maven_checks.xml
ruleset.
[INFO] Ignored 3 errors, 0 violation remaining.
[INFO]
[INFO] --- maven-invoker-plugin:3.2.1:verify (integration-test) @ maven-ear-plugin ---
[INFO] ------------------------------------------------
[INFO] Build Summary:
[INFO]   Passed: 16, Failed: 0, Errors: 0, Skipped: 0
[INFO] ------------------------------------------------
[INFO]
```

```
[INFO] --- maven-failsafe-plugin:2.22.1:verify (default) @ maven-ear-plugin ---
[INFO] Tests are skipped.
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  44.882 s
[INFO] Finished at: 2019-12-03T18:48:53+01:00
[INFO] ------------------------------------------------------------------------
```