

# Integration Testing Framework

## Users Guide

### Table of Contents

1. Overview .....	1
1.1. What is Integration Testing Framework? .....	1
1.2. Status .....	2
2. About this Guide .....	2
3. Overview .....	2
4. Structuring Integration Tests .....	4
4.1. A Single Test Case .....	4
4.2. Test Case Execution .....	5
4.3. Several Test Cases .....	6
5. Goals, Properties and Command Line Options .....	9
5.1. Goals .....	9
5.2. SystemProperties .....	10
5.3. Command Line Options .....	11
6. Scenarios .....	13
6.1. Grouping Test Cases .....	13
6.2. Common Maven Cache .....	14
6.3. Predefined Repository Content for Tests .....	17
6.4. Single Project With Several Executions .....	19
7. Test Case Execution .....	21
7.1. Conditionally Executing Tests .....	22
8. Assertions .....	23
8.1. Overview .....	23
8.2. Expressing Assertions .....	25
9. Configuration Required .....	26
10. Configuration in Maven .....	27

## 1. Overview

This document intends to give a comprehensive reference for programmers writing integration tests for Maven Plugins / Maven Core Extensions / Maven Core.

### 1.1. What is Integration Testing Framework?

The Integration Testing Framework (ITF for short) is in its foundation a [JUnit Jupiter Extension](#)

which supports you in writing integration tests for Maven Plugins etc. There are several aspects that makes writing integration tests for Maven Plugins at the [moment harder than it should be](#). This is the reason why this framework exists.

## 1.2. Status

The current status of this extension is experimental while some people call it Proof of Concept (PoC).

## 2. About this Guide

This guide represents the current state of development and things which work (or more accurate: **should work**). If you find things which do not work as described here or even don't work please don't hesitate to [create an appropriate issue](#) and describe what does not work or does not work at all as described or maybe does not work as you might expect it to work.

### WARNING

This guide is of course not a guarantee that it works cause the project is in a very early stage.

## 3. Overview

The idea of integration tests for Maven Plugins, Maven Extensions, Maven Core is to keep the functionality the way it has been defined independent of refactoring code or improving functionality.

This maven integration test framework is an extension for [JUnit Jupiter](#). The usage of JUnit Jupiter already gives a lot of support for things which are very useful while writing unit- and integration tests. The idea of testing is to express the requirements in code. Those are in other words the tests which should be written.

If you are not familiar with JUnit Jupiter I strongly recommend reading the [JUnit Jupiter User Guide](#) first.

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 test code example which gives you an impression how an integration test for a [Maven Plugins](#)/Maven Extensions/Maven-Core should look like:

```

package org.it;

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

import com.soebes.itf.jupiter.extension.MavenIT;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.maven.MavenExecutionResult;

@MavenIT ①
class FirstMavenIT {

    @MavenTest ②
    void the_first_test_case(MavenExecutionResult result) { ③
        assertThat(result).build().isSuccessful(); ④
    }

}

```

- ① The Maven Integration test annotation
- ② The Maven Test annotation.
- ③ The result of the execution is injected into the test method (details in Chapter [Needs to be written](#))
- ④ The above used assertions like `assertThat(..)` are custom assertions which will be explained in [Assertions chapter](#).

## 4. Structuring Integration Tests

### 4.1. A Single Test Case

The location of an integration test defaults to `src/test/java/<package>/FirstMavenIT.java`. The selected naming schema like `<any>IT.java` implies that it will be executed by the [Maven Failsafe Plugin](#) by convention. This will lead us in a directory structure as follows:

```
.
└── src/
    └── test/
        └── java/
            └── org/
                └── it/
                    └── FirstMavenIT.java
```

In case of an integration test for a Maven plugin/extension or others we need to be able to define also the projects which are the **real test cases** (Maven projects). This needs to be put somewhere in the directory tree to be easily associated with the given test `FirstMavenIT`.

The project to be used as an test case is implied to be located into `src/test/resources-its/<package>/FirstMavenIT` this looks like this:

```
.
└── src/
    └── test/
        └── resources-its/
            └── org/
                └── it/
                    └── FirstMavenIT/
```

Currently this location is separated from all other resources directories to make filtering easier which has to be configured within your `pom.xml` file and preventing interfering with other configurations.

We have an integration test class for example `FirstMavenIT` but what if we like to write several test cases? So we need to make separation between different **test cases** which can be achieved by using the **method name** within the test class `FirstMavenIT` which is `the_first_test_case` in our example. This results in the following directory layout:

```

.
├── src/
│   └── test/
│       ├── resources-its/
│       │   ├── org/
│       │   │   └── it/
│       │   │       ├── FirstMavenIT/
│       │   │       │   └── the_first_test_case/
│       │   │       │       ├── src/
│       │   │       │       └── pom.xml

```

This approach gives us the opportunity to write several integration test cases within a single test class `FirstMavenIT` and also separates them easily. The usage of the **method name** implies some limitations based on the naming rules for **method names**. The best practice is to write **method names** with lowercase letters and separate words by using an underscore `_`. This will prevent issues with case insensitive file systems.

## 4.2. Test Case Execution

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

```

.
├── target/
│   ├── maven-it/
│   │   ├── org/
│   │   │   └── it/
│   │   │       ├── FirstMavenIT/
│   │   │       │   └── the_first_test_case/
│   │   │       │       ├── .m2/
│   │   │       │       ├── project/
│   │   │       │       │   ├── src/
│   │   │       │       │   ├── target/
│   │   │       │       │   └── pom.xml
│   │   │       │       ├── mvn-stdout.log
│   │   │       │       ├── mvn-stderr.log
│   │   │       │       ├── mvn-arguments.log
│   │   │       │       └── orther logs.

```

Based on the above you can see that each **test case** (method within the test class `FirstMavenIT`) has it's own local repository (aka local cache) `.m2/repository`. Furthermore you see that the project is built within the `project` directory. This gives you a view of the built project as you did on plain command line and take a look into it. The output of the build is written into `mvn-stdout.log` (stdout) and the output to stderr is written to `mvn-stderr.log`. The used command line parameters to call Maven are put into `mvn-arguments.log`.

## 4.3. Several Test Cases

If we like to define several integration test cases within a single test class `SeveralMavenIT` we have to define different methods which are the test cases. This results in the following class layout:

```
package org.it;

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

import com.soebes.itf.jupiter.extension.MavenIT;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.maven.MavenExecutionResult;

@MavenIT
class SeveralMavenIT {

    @MavenTest
    void the_first_test_case(MavenExecutionResult result) {
        ...
    }
    @MavenTest
    void the_second_test_case(MavenExecutionResult result) {
        ...
    }
    @MavenTest
    void the_third_test_case(MavenExecutionResult result) {
        ...
    }
}
```

The structure for the Maven projects which are used by each of the test cases (**method names**) looks like the following:

```
.
├── src/
│   └── test/
│       ├── resources-its/
│       │   ├── org/
│       │   │   └── it/
│       │   │       ├── SeveralMavenIT/
│       │   │       │   ├── the_first_test_case/
│       │   │       │   │   ├── src/
│       │   │       │   │   └── pom.xml
│       │   │       │   ├── the_second_test_case/
│       │   │       │   │   ├── src/
│       │   │       │   │   └── pom.xml
│       │   │       │   └── the_this_test_case/
│       │   │       │       ├── src/
│       │   │       │       └── pom.xml
```

After running the integration tests the resulting directory structure in the **target** directory will look like this:

```

.
├── target/
│   ├── maven-it/
│   │   ├── org/
│   │   │   ├── it/
│   │   │   │   ├── SeveralMavenIT/
│   │   │   │   │   ├── the_first_test_case/
│   │   │   │   │   │   ├── .m2/
│   │   │   │   │   │   ├── project/
│   │   │   │   │   │   │   ├── src/
│   │   │   │   │   │   │   ├── target/
│   │   │   │   │   │   │   └── pom.xml
│   │   │   │   │   │   ├── mvn-stdout.log
│   │   │   │   │   │   ├── mvn-stderr.log
│   │   │   │   │   │   └── other logs
│   │   │   │   │   ├── the_second_test_case/
│   │   │   │   │   │   ├── .m2/
│   │   │   │   │   │   ├── project/
│   │   │   │   │   │   │   ├── src/
│   │   │   │   │   │   │   ├── target/
│   │   │   │   │   │   │   └── pom.xml
│   │   │   │   │   │   ├── mvn-stdout.log
│   │   │   │   │   │   ├── mvn-stderr.log
│   │   │   │   │   │   └── other logs
│   │   │   │   │   └── the_third_test_case/
│   │   │   │   │   │   ├── .m2/
│   │   │   │   │   │   ├── project/
│   │   │   │   │   │   │   ├── src/
│   │   │   │   │   │   │   ├── target/
│   │   │   │   │   │   │   └── pom.xml
│   │   │   │   │   │   ├── mvn-stdout.log
│   │   │   │   │   │   ├── mvn-stderr.log
│   │   │   │   │   │   └── mvn-arguments.log

```

Based on the structure you can exactly dive into each test case separately and take a look at the console output of the test case via `mvn-stdout.log` or maybe in case of errors in the `mvn-stderr.log`. In the `project` directory you will find the usual `target` directory which contains the Maven output which might be interesting as well. Furthermore the local cache (aka maven repository) is available separately for each test case and can be found in the `.m2/repository` directory.



# 5. Goals, Properties and Command Line Options

## 5.1. Goals

In each test case method you define `@MavenTest` which says execute Maven with the given default goals and parameters. A typical integration test looks like this:

*BasicIT.java*

```
@MavenIT
class BasicIT {

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

So now the question is: Which goals and parameters will be used to execute Maven for the `first` test case? In general the `@MavenIT` annotation defines a default set of goals which will be executed if not defined otherwise. The default for goals in `@MavenIT` is `package`. That means if we keep the test as in our example maven would be called like `mvn package`. From a technical perspective some other parameters have been added which is `mvn -Dmaven.repo.local=Path package`. The `-Dmaven.repo.local=..` is needed to make sure that each call uses the defined local cache (See [Common Maven Cache](#)). You can of course change the default for the goal if you like by simply changing the parameter for `@MavenIT(goals = {"install"})` that would mean to execute all subjacent tests like `mvn -D.. install` instead of `mvn -D .. package`. A usual command parameter set includes `--batch-mode` and `-V` (This can't be changed currently.).

How could you write a test which uses a plugin goal instead? You can simply define the goal(s) with the `@MavenTest` annotation like this:

```
@MavenTest( goals = {
    "${project.groupId}:${project.artifactId}:${project.version}:compare-dependencies"})
```

The used `goals` in the above `@MavenTest` will overwrite any goal which is defined by `@MavenIT`. The `goals` also supports replacement of placeholders where currently the following are supported:

- `${project.groupId}`
- `${project.artifactId}`
- `${project.version}`

Those are the ones which are used in the majority of cases for Maven plugins. If you like to call several goals and/or lifecycle parts in one go you can simply define it like this:

```
@MavenTest( goals = {
    "${project.groupId}:${project.artifactId}:${project.version}:compare-dependencies",
    "site:stage",
    "install"
})
void test_case(MavenExecutionResult result) {
    ..
}
```

The equivalent on command line would be:

```
mvn ${project.groupId}:${project.artifactId}:${project.version}:compare-dependencies
site:stage install
```

## 5.2. SystemProperties

There are situations where you need to use system properties which are usually defined on command like this:

```
mvn versions:set -DgenerateBackups=false -DnewVersion=2.0
```

This can be achieved by enhancing the `@MavenTest` annotation with `systemProperties` which could look like this:

```

package org.codehaus.mojo.versions.it;

import com.soebes.itf.jupiter.extension.MavenIT;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.maven.MavenExecutionResult;
import com.soebes.itf.jupiter.maven.MavenProjectResult;

import com.soebes.itf.extension.assertj.MavenITAssertions.assertThat;

@MavenIT
class CompareDependenciesIT
{
    @MavenTest( goals = {
        "${project.groupId}:${project.artifactId}:${project.version}:compare-dependencies"},
        systemProperties = {
            "remotePom=localhost:dummy-bom-pom:1.0",
            "reportOutputFile=target/depDiffs.txt"
        }
    )
    void it_compare_dependencies_001( MavenExecutionResult result )
    {
        ...
    }
}

```

## 5.3. Command Line Options

In different scenarios it is needed to define command line options for example `--non-recursive` etc. This can be done by using the `options` part of `@MavenTest`. There is a convenience class `MavenOptions` available which contains all existing command line options. You are not forced to use the `MavenOptions` class.

```

@MavenTest( options = {MavenOptions.NON_RECURSIVE, "--offline"},
    goals = {"${project.groupId}:${project.artifactId}:${project.version}:set"},
    systemProperties = {"newVersion=2.0"} )
void first( MavenExecutionResult result )
{
    assertThat( result ).isSuccessful();
}

```

This gives you the choice to decide to use `MavenOptions` or not:

```
@MavenTest( options = {"-N", "--offline"},
            goals = {"${project.groupId}:${project.artifactId}:${project.version}:set"},
            systemProperties = {"newVersion=2.0"} )
void first( MavenExecutionResult result )
{
    assertThat( result ).isSuccessful();
}
```

## 6. Scenarios

### 6.1. Grouping Test Cases

Sometimes it makes sense to group test into different groups together. This can be achieved via the `@Nested` annotation which is provided by [JUnit Jupiter](#). This would result in a test class like this:

*MavenIntegrationGroupingIT.java*

```
@MavenIT
class MavenIntegrationGroupingIT {

    @MavenTest
    void packaging_includes(MavenExecutionResult result) {
    }

    @Nested
    class NestedExample {

        @MavenTest
        void basic(MavenExecutionResult result) {
        }

        @MavenTest
        void packaging_includes(MavenExecutionResult result) {
        }

    }
}
```

After test execution the resulting directory tree looks like this:

```

.
├── target/
│   ├── maven-it/
│   │   ├── org/
│   │   │   ├── it/
│   │   │   │   ├── MavenIntegrationGroupingIT/
│   │   │   │   │   ├── packaging_includes/
│   │   │   │   │   │   ├── .m2/
│   │   │   │   │   │   ├── project/
│   │   │   │   │   │   │   ├── src/
│   │   │   │   │   │   │   ├── target/
│   │   │   │   │   │   │   └── pom.xml
│   │   │   │   │   │   ├── mvn-stdout.log
│   │   │   │   │   │   ├── mvn-stderr.log
│   │   │   │   │   │   └── other logs
│   │   │   │   │   └── NestedExample/
│   │   │   │   │   │   ├── basic/
│   │   │   │   │   │   │   ├── .m2/
│   │   │   │   │   │   │   ├── project/
│   │   │   │   │   │   │   │   ├── src/
│   │   │   │   │   │   │   │   ├── target/
│   │   │   │   │   │   │   │   └── pom.xml
│   │   │   │   │   │   │   ├── mvn-stdout.log
│   │   │   │   │   │   │   ├── mvn-stderr.log
│   │   │   │   │   │   │   └── other logs
│   │   │   │   │   │   └── packaging_includes/
│   │   │   │   │   │   │   ├── .m2/
│   │   │   │   │   │   │   ├── project/
│   │   │   │   │   │   │   │   ├── src/
│   │   │   │   │   │   │   │   ├── target/
│   │   │   │   │   │   │   │   └── pom.xml
│   │   │   │   │   │   │   ├── mvn-stdout.log
│   │   │   │   │   │   │   ├── mvn-stderr.log
│   │   │   │   │   │   │   └── other logs

```

## 6.2. Common Maven Cache

In all previous test case examples the maven cache (aka maven repository) is created separately for each of the test cases (**test methods**). There are times where you need to have a common cache (aka maven repository) for two or more test cases together. This can be achieved easily via the `@MavenRepository` annotation.<sup>[1]</sup> The usage looks like the following:

```
package org.it;

import com.soebes.itf.jupiter.extension.MavenIT;
import com.soebes.itf.jupiter.extension.MavenRepository;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.maven.MavenExecutionResult;

@MavenIT
@MavenRepository
class MavenITWithGlobalMavenCacheIT {

    @MavenTest
    void packaging_includes(MavenExecutionResult result) {
    }

    @MavenTest
    void basic(MavenExecutionResult result) {
    }

}
```

After test execution the resulting directory tree looks like this:

```
.
├── target/
│   ├── maven-it/
│   │   ├── org/
│   │   │   ├── it/
│   │   │   │   ├── MavenITWithGlobalMavenCacheIT/
│   │   │   │   │   ├── .m2/
│   │   │   │   │   ├── packaging_includes/
│   │   │   │   │   │   ├── project/
│   │   │   │   │   │   │   ├── src/
│   │   │   │   │   │   │   ├── target/
│   │   │   │   │   │   │   └── pom.xml
│   │   │   │   │   │   ├── mvn-stdout.log
│   │   │   │   │   │   ├── mvn-stderr.log
│   │   │   │   │   │   └── other logs
│   │   │   │   │   └── basic/
│   │   │   │   │       ├── project/
│   │   │   │   │       │   ├── src/
│   │   │   │   │       │   ├── target/
│   │   │   │   │       │   └── pom.xml
│   │   │   │   │       ├── mvn-stdout.log
│   │   │   │   │       ├── mvn-stderr.log
│   │   │   │   │       └── other logs
```

There you see that the `.m2/` directory (maven local cache) is directly located under the `MavenITWithGlobalMavenCacheIT` directory which is the equivalent of the `MavenITWithGlobalMavenCacheIT` class.

The usage of `@MavenRepository` is also possible in combination with `@Nested` annotation which will look like this:

*MavenIntegrationGroupingIT.java*

```
@MavenIT
class MavenIntegrationGroupingIT {

    @MavenTest
    void packaging_includes(MavenExecutionResult result) {
    }

    @Nested
    @MavenRepository
    class NestedExample {

        @MavenTest
        void basic(MavenExecutionResult result) {
        }

        @MavenTest
        void packaging_excludes(MavenExecutionResult result) {
        }

    }
}
```

That would result in having a common cache for the methods `basic` and `packaging_includes` within the nested class `NestedExample`. The test method `packaging_includes` will have a cache on it's own. The directory tree looks like this:



```

.
├── target/
│   ├── maven-it/
│   │   ├── org/
│   │   │   ├── it/
│   │   │   │   ├── MavenIntegrationGroupingIT/
│   │   │   │   │   ├── packaging_includes/
│   │   │   │   │   │   ├── .m2/
│   │   │   │   │   │   │   ├── project/
│   │   │   │   │   │   │   │   ├── src/
│   │   │   │   │   │   │   │   ├── target/
│   │   │   │   │   │   │   │   └── pom.xml
│   │   │   │   │   │   │   ├── mvn-stdout.log
│   │   │   │   │   │   │   ├── mvn-stderr.log
│   │   │   │   │   │   │   └── other logs
│   │   │   │   │   └── NestedExample/
│   │   │   │   │   │   ├── .m2/
│   │   │   │   │   │   │   ├── basic/
│   │   │   │   │   │   │   │   ├── project/
│   │   │   │   │   │   │   │   │   ├── src/
│   │   │   │   │   │   │   │   │   ├── target/
│   │   │   │   │   │   │   │   │   └── pom.xml
│   │   │   │   │   │   │   │   ├── mvn-stdout.log
│   │   │   │   │   │   │   │   ├── mvn-stderr.log
│   │   │   │   │   │   │   │   └── other logs
│   │   │   │   │   │   └── packaging_excludes/
│   │   │   │   │   │   │   ├── project/
│   │   │   │   │   │   │   │   ├── src/
│   │   │   │   │   │   │   │   ├── target/
│   │   │   │   │   │   │   │   └── pom.xml
│   │   │   │   │   │   │   ├── mvn-stdout.log
│   │   │   │   │   │   │   ├── mvn-stderr.log
│   │   │   │   │   │   │   └── other logs

```

## 6.3. Predefined Repository Content for Tests

There are existing test cases in which you need predefined dependencies cause you can't rely on existing dependencies in the central repository or anywhere else. You have the option to define a repository either per testcase or per test class where you can put existing dependencies for your test cases. Those dependencies are behaving like you have installed in your local repository via `mvn install:install-file`.

In your test code the setup looks like the following. The important part is the definition of the `@MavenPredefinedRepository` which indicated that the predefined repository in the appropriate location needs to exist. This means that those dependencies from this directory are available for each test case (`project_001` and `proeject_002`) in this test class.

```
@MavenIT
@MavenPredefinedRepository
class ProjectIT {

    @MavenTest(options = {MavenOptions.DEBUG})
    void project_001(MavenExecutionResult result) {
        assertThat(result).isSuccessful();
    }

    @MavenTest(options = {MavenOptions.DEBUG})
    void project_002(MavenExecutionResult result) {
        assertThat(result).isSuccessful();
    }

}
```

In your test project definition you need to create the appropriate directory structure including the needed content which looks like this:

```
.
├── src/
│   ├── test/
│   │   ├── resources-its/
│   │   │   ├── org/
│   │   │   │   ├── it/
│   │   │   │   │   ├── ProjectIT/
│   │   │   │   │   │   ├── .predefined-repo
│   │   │   │   │   │   │   ├── ...
│   │   │   │   │   │   │   └── ...
│   │   │   │   │   ├── project_001/
│   │   │   │   │   │   ├── src/
│   │   │   │   │   │   └── pom.xml
│   │   │   │   │   ├── project_002/
│   │   │   │   │   │   ├── src/
│   │   │   │   │   │   └── pom.xml
```

In the `.predefined-repo` you have to follow a usual maven repository structure. You can of course define the `@MavenPredefinedRepository` also on the test method level which would look like this:

```

MavenIT
class ProjectIT {

    @MavenTest(options = {MavenOptions.DEBUG})
    @MavenPredefinedRepository
    void project_001(MavenExecutionResult result) {
        assertThat(result).isSuccessful();
    }

    @MavenTest(options = {MavenOptions.DEBUG})
    @MavenPredefinedRepository
    void project_002(MavenExecutionResult result) {
        assertThat(result).isSuccessful();
    }

}

```

The setup directories look like this:

```

.
├── src/
│   ├── test/
│   │   ├── resources-its/
│   │   │   ├── org/
│   │   │   │   ├── it/
│   │   │   │   │   ├── ProjectIT/
│   │   │   │   │   │   ├── project_001/
│   │   │   │   │   │   │   ├── .predefined-repo
│   │   │   │   │   │   │   ├── src/
│   │   │   │   │   │   │   └── pom.xml
│   │   │   │   │   │   └── project_002/
│   │   │   │   │   │       ├── .predefined-repo
│   │   │   │   │   │       ├── src/
│   │   │   │   │   │       └── pom.xml

```

## 6.4. Single Project With Several Executions

Sometimes you need to execute a consecutive number of commands (usually maven executions) on the same single project. This means in the end having a single project and executing several maven execution on that project. Such a use case looks like this:

```

MavenIT
class SetIT
{
    private static final String VERSIONS_PLUGIN =
        "${project.groupId}:${project.artifactId}:${project.version}";

    @Nested
    @MavenProject
    @TestMethodOrder( OrderAnnotation.class )
    class set_001
    {

        @MavenTest(
            options = MavenOptions.NON_RECURSIVE,
            goals = {VERSIONS_PLUGIN + ":set"},
            systemProperties = {"newVersion=2.0"} )
        @Order(10)
        void first( MavenExecutionResult result )
        {
            assertThat( result ).isSuccessful();
        }

        @MavenTest(
            options = MavenOptions.NON_RECURSIVE,
            goals = {VERSIONS_PLUGIN + ":set"},
            systemProperties = {
                "newVersion=2.0",
                "groupId=",
                "artifactId=",
                "oldVersion="} )
        @Order(20)
        void second( MavenExecutionResult result)
        {
            assertThat( result ).isSuccessful();
        }
    }
}

```

The important part here is the `@MavenProject` annotation which marks the nested class as a container which contains executions (`first` and `second`) with conditions on the same single project. The `@MavenProject` defines that project name which is by default `maven_project`. This means you have to define the project you would like to test on like this:

```

.
├── src/
│   └── test/
│       ├── resources-its/
│       │   ├── org/
│       │   │   ├── it/
│       │   │   │   ├── SetIT/
│       │   │   │   │   ├── set_001/
│       │   │   │   │   │   ├── maven_project/
│       │   │   │   │   │   │   ├── src/
│       │   │   │   │   │   │   └── pom.xml

```

After test execution it looks like this:

```

.
├── target/
│   ├── maven-it/
│   │   ├── org/
│   │   │   ├── it/
│   │   │   │   ├── SetIT/
│   │   │   │   │   ├── set_001/
│   │   │   │   │   │   ├── maven_project/
│   │   │   │   │   │   │   ├── .m2/
│   │   │   │   │   │   │   ├── project/
│   │   │   │   │   │   │   │   ├── src/
│   │   │   │   │   │   │   │   ├── target/
│   │   │   │   │   │   │   └── pom.xml
│   │   │   │   │   │   ├── first-mvn-arguments.log
│   │   │   │   │   │   ├── first-mvn-stdout.log
│   │   │   │   │   │   ├── first-mvn-stderr.log
│   │   │   │   │   │   ├── second-mvn-arguments.log
│   │   │   │   │   │   ├── second-mvn-stdout.log
│   │   │   │   │   │   └── second-mvn-stderr.log

```

Each test case defined by the method name **first** and **second** has been executed on the same project **maven\_project**. Each execution has its own sets of log files which can be identified by the prefix based on the method name like **first-mvn-arguments.log** etc.

The **@MavenProject** annotation can only be used on a nested class or on the test class itself (where **@MavenIT** is located.). If you like to change the name of the project **maven\_project** into something different this can be achieved by using **@MavenProject("another\_project\_name")**.

## 7. Test Case Execution

## 7.1. Conditionally Executing Tests

You might want to run an integration test only for a particular Maven version for example running only for Maven 3.6.0? So how could you express this? The following code will show how you can do that.

*ForthMavenIT.java*

```
import static com.soebes.itf.extension.assertj.MavenCacheResultAssert.assertThat;
import static com.soebes.itf.jupiter.maven.MavenVersion.M3_0_5;
import static com.soebes.itf.jupiter.maven.MavenVersion.M3_6_0;

import com.soebes.itf.jupiter.extension.DisabledForMavenVersion;
import com.soebes.itf.jupiter.extension.EnabledForMavenVersion;
import com.soebes.itf.jupiter.extension.MavenIT;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.maven.MavenExecutionResult;

@MavenIT
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();
    }

}
```

If you like to disable some tests on a particular Java version this can be handled via [conditions like this](#).

```

import static com.soebes.itf.extension.assertj.MavenITAssertions.assertThat;
import static com.soebes.itf.jupiter.maven.MavenVersion.M3_0_5;
import static com.soebes.itf.jupiter.maven.MavenVersion.M3_6_0;

import com.soebes.itf.jupiter.extension.DisabledForMavenVersion;
import com.soebes.itf.jupiter.extension.EnabledForMavenVersion;
import com.soebes.itf.jupiter.extension.MavenIT;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.maven.MavenExecutionResult;
import org.junit.jupiter.api.condition.DisabledOnJre;
import org.junit.jupiter.api.condition.JRE;

@MavenIT
@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();
    }
}

```

## 8. Assertions

### 8.1. Overview

Let us take a look into a simple integration test. We would like to concentrate on the assertion part.

```

@MavenIT
class FirstIT {
    @MavenTest
    void the_first_test_case(MavenExecutionResult result) {
        assertThat(result).isSuccessful();
    }
}

```

After the test has run the resulting directory structure looks like this:

```

.
├── target/
│   ├── maven-its/
│   │   ├── org/
│   │   │   ├── it/
│   │   │   │   ├── FirstIT/
│   │   │   │   │   ├── the_first_test_case/
│   │   │   │   │   │   ├── .m2/
│   │   │   │   │   │   ├── project/
│   │   │   │   │   │   │   ├── src/
│   │   │   │   │   │   │   ├── target/
│   │   │   │   │   │   │   └── pom.xml
│   │   │   │   │   ├── mvn-stdout.log
│   │   │   │   │   ├── mvn-stderr.log
│   │   │   │   │   └── other logs

```

In each integration test you should let inject `MavenExecutionResult result` as a parameter of your test case method cause that gives you the opportunity to write assertion on the result of the maven execution or what has written into the resulting structure.

Let us start with two general assertions:

- `assertThat(result).isSuccessful();` The build was successful (return code of Maven run 0).
- `assertThat(result).isFailure();` The build has failed (return code of Maven run != 0).

Sometimes this is sufficient but more often you have more complex scenarios to be checked.

Based on the directory structure in the result you can make assumptions about the names which can be used in your assertions like the following:

- `assertThat(result).project()...` which will go into the `project` directory
- `assertThat(result).cache()...` will go into the `.m2/repository` directory.
- `assertThat(result).log()..'` will result into the access of the `'mvn-stdout.log` file.

So next will be to check that a file in the `target` directory has been created during a test and should contain the required content. How should that be expressed? The following gives you an example how you can achieve that:



```

assertThat( result ).isSuccessful()
    .project()
    .hasTarget()
        .withFile( "depDiffs.txt" )
            .hasContent( String.join( "\n",
                "The following differences were found:",
                "",
                "    none", "",
                "The following property differences were found:",
                "",
                "    none" ) );

```

The first part `.isSuccessful()` checks that the build has gone fine than we go into `project` directory and via `withTarget()` we check the existence of the `target` directory as well as going into that directory. Finally we append `withFile(...)` which selects which file and redirects to the `AbstractFileAssert<?>` of AssertJ which gives you the choice to check the content of the file as you like.

```

assertThat(project).hasTarget()
    .withEarFile()
    .containsOnlyOnce("META-INF/application.xml", "META-INF/appserver-application.xml");

```

## 8.2. Expressing Assertions

```

package org.codehaus.mojo.versions.it;

import com.soebes.itf.jupiter.extension.MavenIT;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.maven.MavenExecutionResult;
import com.soebes.itf.jupiter.maven.MavenProjectResult;

import com.soebes.itf.extension.assertj.MavenITAssertions.assertThat;

@MavenIT
class CompareDependenciesIT
{
    @MavenTest(
        goals =
            {"${project.groupId}:${project.artifactId}:${project.version}:compare-
dependencies"},
        systemProperties = {
            "remotePom=localhost:dummy-bom-pom:1.0",
            "reportOutputFile=target/depDiffs.txt"
        } )
    void it_compare_dependencies_001( MavenExecutionResult result )
    {
        assertThat( result ).isSuccessful()
            .project()
            .hasTarget()
            .withFile( "depDiffs.txt" )
            .hasContent( String.join( "\n",
                "The following differences were found:",
                "",
                "  none",
                "",
                "The following property differences were found:",
                "",
                "  none" ) );
    }
}

```

## 9. Configuration Required

If you like to use the integration test framework you would like to use it to write tests and run those tests from command line (via Maven). This requires that the configuration is done as described in [Configuration in Maven](#)

If you like to run your integration tests from you IDE you have to have configured that the maven binary can be found via the **PATH**.

You can overwrite the **maven.home** property to control which Maven version will be used during the

tests. Take care with overwriting properties in Maven.

## 10. Configuration in Maven

You have decided to use this tool to test your plugin/extension etc. this means you have to build it on your own. I strongly recommend to start with most recent release version which makes it easier for you. If you like to use the most recent development version you can of course build it on your own.

**NOTE** Starting with version 0.2.0 it will be published to central repository.

### Prerequisites

- JDK 8+ must be installed
- [Apache Maven 3.6.3](#)

You have to call the following command line to build it:

```
mvn clean install
```

This will run the whole build including the execution of the integration test in the `itf-examples` modules. This can take a while and should produce an output similar like this:

```
[INFO]
[INFO] --- maven-jar-plugin:3.1.1:jar (default-jar) @ itf-examples ---
[WARNING] JAR will be empty - no content was marked for inclusion!
[INFO] Building jar: /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-examples/target/itf-examples-0.2.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-site-plugin:3.7.1:attach-descriptor (attach-descriptor) @ itf-examples ---
[INFO] Skipping because packaging 'jar' is not pom.
[INFO]
[INFO] --- maven-invoker-plugin:3.2.1:install (integration-test) @ itf-examples ---
[INFO] Installing /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-jupiter-extension/pom.xml to /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-examples/target/itf-repo/com/soebes/itf/jupiter/extension/itf-jupiter-extension/0.2.0-SNAPSHOT/itf-jupiter-extension-0.2.0-SNAPSHOT.pom
[INFO] Installing /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-jupiter-extension/target/itf-jupiter-extension-0.2.0-SNAPSHOT.jar to /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-examples/target/itf-repo/com/soebes/itf/jupiter/extension/itf-jupiter-extension/0.2.0-SNAPSHOT/itf-jupiter-extension-0.2.0-SNAPSHOT.jar
[INFO] Installing /Users/khmarbaise/ws-git-apache/maven-it-extension/pom.xml to /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-examples/target/itf-repo/com/soebes/itf/jupiter/extension/itf-extension/0.2.0-SNAPSHOT/itf-extension-0.2.0-SNAPSHOT.pom
[INFO] Installing /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-extension-
```

```

maven/pom.xml to /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-
examples/target/itf-repo/com/soebes/itf/jupiter/extension/itf-extension-maven/0.2.0-
SNAPSHOT/itf-extension-maven-0.2.0-SNAPSHOT.pom
[INFO] Installing /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-extension-
maven/target/itf-extension-maven-0.2.0-SNAPSHOT.jar to /Users/khmarbaise/ws-git-
apache/maven-it-extension/itf-examples/target/itf-
repo/com/soebes/itf/jupiter/extension/itf-extension-maven/0.2.0-SNAPSHOT/itf-
extension-maven-0.2.0-SNAPSHOT.jar
[INFO] Installing /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-
assertj/pom.xml to /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-
examples/target/itf-repo/com/soebes/itf/jupiter/extension/itf-assertj/0.2.0-
SNAPSHOT/itf-assertj-0.2.0-SNAPSHOT.pom
[INFO] Installing /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-
assertj/target/itf-assertj-0.2.0-SNAPSHOT.jar to /Users/khmarbaise/ws-git-
apache/maven-it-extension/itf-examples/target/itf-
repo/com/soebes/itf/jupiter/extension/itf-assertj/0.2.0-SNAPSHOT/itf-assertj-0.2.0-
SNAPSHOT.jar
[INFO] Installing /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-
examples/pom.xml to /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-
examples/target/itf-repo/com/soebes/itf/jupiter/extension/itf-examples/0.2.0-
SNAPSHOT/itf-examples-0.2.0-SNAPSHOT.pom
[INFO] Installing /Users/khmarbaise/ws-git-apache/maven-it-extension/itf-
examples/target/itf-examples-0.2.0-SNAPSHOT.jar to /Users/khmarbaise/ws-git-
apache/maven-it-extension/itf-examples/target/itf-
repo/com/soebes/itf/jupiter/extension/itf-examples/0.2.0-SNAPSHOT/itf-examples-0.2.0-
SNAPSHOT.jar
[INFO]
[INFO] --- maven-failsafe-plugin:3.0.0-M4:integration-test (default) @ itf-examples
---
[INFO]
[INFO] -----
[INFO]  T E S T S
[INFO] -----
[INFO] Running com.soebes.itf.examples.EARIT
[INFO] Running com.soebes.itf.examples.MavenProjectIT
[INFO] Running com.soebes.itf.examples.MavenIntegrationExampleNestedIT
[INFO] Running com.soebes.itf.examples.ITWithBeforeEachIT
[INFO] Running
com.soebes.itf.examples.MavenIntegrationExampleNestedWithoutNoneNestedTestGlobalRepoIT
[INFO] Running com.soebes.itf.examples.MavenIntegrationIT
[INFO] Running com.soebes.itf.examples.MavenIntegrationExampleNestedIT$NestedExample
[INFO] Running
com.soebes.itf.examples.MavenIntegrationExampleNestedWithoutNoneNestedTestGlobalRepoIT
$NestedExample
[INFO] Running com.soebes.itf.examples.MavenProjectIT$NestedExample
* beforeEach of ITWithBeforeEachIT
(basic) result = MavenExecutionResult[result=Successful, returnCode=0,
mavenLog=com.soebes.itf.jupiter.maven.MavenLog@18b0d71,
mavenProjectResult=com.soebes.itf.jupiter.maven.MavenProjectResult@6ba7b73,
mavenCacheResult=com.soebes.itf.jupiter.maven.MavenCacheResult@50d1e210]
result = MavenExecutionResult[result=Successful, returnCode=0,

```

```

mavenLog=com.soebes.itf.jupiter.maven.MavenLog@b045628,
mavenProjectResult=com.soebes.itf.jupiter.maven.MavenProjectResult@49ecd06a,
mavenCacheResult=com.soebes.itf.jupiter.maven.MavenCacheResult@1994110a]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 11.325 s - in
com.soebes.itf.examples.ITWithBeforeEachIT
(packaging_includes) result = MavenExecutionResult[result=Successful, returnCode=0,
mavenLog=com.soebes.itf.jupiter.maven.MavenLog@464cc2cb,
mavenProjectResult=com.soebes.itf.jupiter.maven.MavenProjectResult@459feafd,
mavenCacheResult=com.soebes.itf.jupiter.maven.MavenCacheResult@5484e35a]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 17.442 s - in
com.soebes.itf.examples.MavenProjectIT$NestedExample
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 18.279 s - in
com.soebes.itf.examples.MavenIntegrationExampleNestedWithoutNoneNestedTestGlobalRepoIT
$NestedExample
[INFO] Tests run: 0, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 18.297 s - in
com.soebes.itf.examples.MavenIntegrationExampleNestedWithoutNoneNestedTestGlobalRepoIT
[INFO] Running com.soebes.itf.examples.ITWithBeforeEachMavenIT
* beforeEach of ITWithBeforeEachIT
[INFO] Running com.soebes.itf.examples.OptionsIT
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 21.473 s - in
com.soebes.itf.examples.MavenIntegrationExampleNestedIT$NestedExample
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 27.352 s - in
com.soebes.itf.examples.MavenProjectIT
result = MavenExecutionResult[result=Successful, returnCode=0,
mavenLog=com.soebes.itf.jupiter.maven.MavenLog@2cf8d855,
mavenProjectResult=com.soebes.itf.jupiter.maven.MavenProjectResult@62d887cc,
mavenCacheResult=com.soebes.itf.jupiter.maven.MavenCacheResult@6ee151ef]
[WARNING] Tests run: 2, Failures: 0, Errors: 0, Skipped: 1, Time elapsed: 9.787 s - in
com.soebes.itf.examples.ITWithBeforeEachMavenIT
[WARNING] Tests run: 5, Failures: 0, Errors: 0, Skipped: 2, Time elapsed: 31.526 s -
in com.soebes.itf.examples.MavenIntegrationExampleNestedIT
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 14.384 s - in
com.soebes.itf.examples.OptionsIT
MavenIntegrationIT.first_integration_test rc:0
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 38.953 s - in
com.soebes.itf.examples.MavenIntegrationIT
[INFO] Running com.soebes.itf.examples.MavenProjectRootIT
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 40.687 s - in
com.soebes.itf.examples.EARIT
(basic) result = MavenExecutionResult[result=Successful, returnCode=0,
mavenLog=com.soebes.itf.jupiter.maven.MavenLog@64eeb421,
mavenProjectResult=com.soebes.itf.jupiter.maven.MavenProjectResult@6e5f8454,
mavenCacheResult=com.soebes.itf.jupiter.maven.MavenCacheResult@69c72cf9]
(packaging_includes) result = MavenExecutionResult[result=Successful, returnCode=0,
mavenLog=com.soebes.itf.jupiter.maven.MavenLog@7c6516ff,
mavenProjectResult=com.soebes.itf.jupiter.maven.MavenProjectResult@6d3bfaee,
mavenCacheResult=com.soebes.itf.jupiter.maven.MavenCacheResult@7f54be39]
(basic) result = MavenExecutionResult[result=Successful, returnCode=0,
mavenLog=com.soebes.itf.jupiter.maven.MavenLog@2f24e5ce,
mavenProjectResult=com.soebes.itf.jupiter.maven.MavenProjectResult@7b22dc3b,
mavenCacheResult=com.soebes.itf.jupiter.maven.MavenCacheResult@49fca53b]

```

```

[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 8.494 s - in
com.soebes.itf.examples.MavenProjectRootIT
[INFO] Running com.soebes.itf.examples.MavenIntegrationExampleNestedGlobalRepoIT
beforeEachOne: result = null
beforeEachOne: MavenIntegrationExampleNestedGlobalRepoIT.beforeEach
beforeEachTwo: result = null
beforeEachTwo: MavenIntegrationExampleNestedGlobalRepoIT.beforeEach
[INFO] Running
com.soebes.itf.examples.MavenIntegrationExampleNestedGlobalRepoIT$NestedExample
beforeEachOne: result = null
beforeEachOne: MavenIntegrationExampleNestedGlobalRepoIT.beforeEach
beforeEachTwo: result = null
beforeEachTwo: MavenIntegrationExampleNestedGlobalRepoIT.beforeEach
beforeEachOne: result = null
beforeEachOne: MavenIntegrationExampleNestedGlobalRepoIT.beforeEach
beforeEachTwo: result = null
beforeEachTwo: MavenIntegrationExampleNestedGlobalRepoIT.beforeEach
[WARNING] Tests run: 4, Failures: 0, Errors: 0, Skipped: 1, Time elapsed: 3.455 s - in
com.soebes.itf.examples.MavenIntegrationExampleNestedGlobalRepoIT$NestedExample
[INFO] Tests run: 0, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 7.401 s - in
com.soebes.itf.examples.MavenIntegrationExampleNestedGlobalRepoIT
[INFO]
[INFO] Results:
[INFO]
[WARNING] Tests run: 31, Failures: 0, Errors: 0, Skipped: 4
[INFO]
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.4:report (default) @ itf-examples ---
[INFO] Skipping JaCoCo execution due to missing classes directory.
[INFO]
[INFO] --- maven-failsafe-plugin:3.0.0-M4:verify (default) @ itf-examples ---
[INFO] -----
[INFO] Reactor Summary for itf-extension 0.2.0-SNAPSHOT:
[INFO]
[INFO] itf-extension ..... SUCCESS [ 1.215 s]
[INFO] itf-executor ..... SUCCESS [ 0.417 s]
[INFO] itf-maven-plugin ..... SUCCESS [ 0.023 s]
[INFO] itf-extension-maven ..... SUCCESS [ 0.448 s]
[INFO] itf-jupiter-extension ..... SUCCESS [ 4.365 s]
[INFO] itf-assertj ..... SUCCESS [ 1.390 s]
[INFO] itf-examples ..... SUCCESS [01:00 min]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 01:09 min
[INFO] Finished at: 2020-04-15T20:53:51+02:00
[INFO] -----

```

If you like to know how a start point looks like you can take a look into the [itf-examples](#) project which shows the current configuration you need to have.

The first part are the dependencies you need to add to your project. The `${project.version}` has to be replaced with the current version of the compiled project currently for example: `0.1.0-SNAPSHOT`. The `assertj-core` dependency is there cause I have used for development and currently nothing else is supported(see `itf-assertj`).

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.soebes.itf.jupiter.extension</groupId>
  <artifactId>itf-assertj</artifactId>
  <version>${project.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.soebes.itf.jupiter.extension</groupId>
  <artifactId>itf-jupiter-extension</artifactId>
  <version>${project.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-core</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
</dependency>
```

Further more you have to copy the `src/test/resources-its` directory including filtering. You need to be careful which delimiter you have chosen. So you might need to tweak your configuration based on your choices.

```

<testResources>
  <testResource>
    <directory>src/test/resources</directory>
    <filtering>>false</filtering>
  </testResource>
  <!--
    ! Currently those tests do need to be filtered.
  -->
  <testResource>
    <directory>src/test/resources-its</directory>
    <filtering>>true</filtering>
  </testResource>
</testResources>
</plugins>

```

The next thing is you have to configure `itf-maven-plugin` with the `install` goal like this:

```

<plugin>
  <groupId>com.soebes.itf.jupiter.extension</groupId>
  <artifactId>itf-maven-plugin</artifactId>
  <version>${project.version}</version>
  <executions>
    <execution>
      <id>installing</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>install</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

Finally you have to add a configuration for [Maven Failsafe Plugin](#) like the following:



```

<plugin>
  <artifactId>maven-failsafe-plugin</artifactId>
  <configuration>
    <!--
      ! currently needed to run integration tests.
    -->
    <systemProperties>
      <maven.version>${maven.version}</maven.version>
      <maven.home>${maven.home}</maven.home>
    </systemProperties>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

The two properties given `maven.version` transfers the version of Maven which is used within the itf-jupiter-extension and the `maven.home` transfers the information of where to find the current Maven installation. This is needed to start the Maven process from within the integration tests.

The usage of [Maven Failsafe Plugin](#) implies the naming convention for the integration tests like `*IT.java` but of course you can change that by using the appropriate configuration.

#### NOTE

The whole given configuration should be replaced by separate maven plugin later. The first part meaning the installation of the artifacts is done but of course there are parts which where missing.

[1] Base on the usage of this annotation the parallelizing is automatically deactivated cause Maven has never been designed to make a parallel access to the maven cache possible.