# Integration Testing Framework Users Guide

## Table of Contents

# 1. Overview

The documentation 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). This framework has not yet reached version `1.0.0` which means that anything can change, but I try to keep compatibility as much as possible. I will only break it if really needed. In such cases it will be documented in the release notes. I strongly recommend reading the release notes.

# 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 at all, 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 significance 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 com.soebes.itf.extension.assertj.MavenITAssertions.assertThat;

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

@MavenJupiterExtension ①
public class FirstMavenIT {

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

}
```

① The Maven Integration test annotation

② The Maven Test annotation.

③ The result of the execution 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. Configuration in Maven

## 4.1. User's Point of View

You are a user who want to use the integration test framework to write integration tests for a plugin etc. This area shows how to configure the integration test framework in your Maven build and what kind of requirements exist.

The prerequisites to use this integration test framework is, that you are running JDK8 at minimum for your tests. This is based on using JUnit Jupiter Extension and of course on the implemented code of this extension.

The requirements are:

- JDK8+
- Apache Maven 3.1.0 or above.

The first step is to add the appropriate dependencies to your project. They are usually with `test` scope, cause you only need them during the integration tests.

```
<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-core</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.soebes.itf.jupiter.extension</groupId>
  <artifactId>itf-assertj</artifactId>
  <version>0.14.0-SNAPSHOT</version>
  <scope>test</scope>
</dependency>
<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-jupiter-extension</artifactId>
  <version>0.14.0-SNAPSHOT</version>
  <scope>test</scope>
</dependency>
```

The dependency `com.soebes.itf.jupiter.extension:itf-assertj` contains custom assertions of AssertJ in case you want to use AssertJ as your assertion framework. This means you have to include `org.assertj:assertj-core` as well. If you don't want to use AssertJ as assertion framework you can omit them both.

Now you have to have the dependency `org.junit.jupiter:junit-jupiter-engine` to get tests running with JUnit Jupiter and finally you have to add the `com.soebes.itf.jupiter.extension:itf-jupiter-extension` dependency to get the support for your Maven integration tests.

Based on the described structures the content (with the projects which is used as test) from `src/test/resources-its` has to be copied into `target/test-classes` which usually means including filtering. This is used to replace placeholders in files like `@project.version@` and replace with the real version of your extension/plugin etc. so you can use the `itf-maven-plugin` which is doing this job on its own which means you don't have to do something special here.

The next thing is to configure the `itf-maven-plugin` with the `install` and the `resources-its` goal like this:

```
        <plugin>
          <groupId>com.soebes.itf.jupiter.extension</groupId>
          <artifactId>itf-maven-plugin</artifactId>
          <version>0.14.0-SNAPSHOT</version>
          <executions>
            <execution>
              <id>installing</id>
              <phase>pre-integration-test</phase>
              <goals>
                <goal>install</goal>
                <goal>resources-its</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
```

The `itf-maven-plugin` copies the code of your extension/plugin into appropriate directories which are used during the integration tests. Further more it will handle the copying of the structure from `src/test/resources-its` into the correct location and will also handle the filtering as described above.

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.
            -->
            <systemPropertyVariables>
              <maven.version>${maven.version}</maven.version>
              <maven.home>${maven.home}</maven.home>
            </systemPropertyVariables>
            <properties>
              <configurationParameters>
                junit.jupiter.execution.parallel.enabled=true
```

```
                junit.jupiter.execution.parallel.mode.default=concurrent
                junit.jupiter.execution.parallel.mode.classes.default=same_thread
                junit.jupiter.execution.parallel.config.strategy=fixed
                junit.jupiter.execution.parallel.config.fixed.parallelism=12
              </configurationParameters>
            </properties>
          </configuration>
          <executions>
            <execution>
              <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
```

The given properties like `maven.version` transfers the version of Maven which is used within the `itf-jupiter-extension` to run your integration tests 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 if you like or need to.

The above described configuration will make it possible to run all tests via command line by using `mvn clean verify`.

| | |
|---|---|
| **IMPORTANT** | It is also possible to run the integration tests from your IDE. This only works if you have done the `mvn clean verify` once before on command line. Then you can run the integration tests from your IDE as long as you don't change the code of the plugin/extension etc. which is under test. You can change the tests without any limitation (If you find any please create an issue for that.) |

| | |
|---|---|
| **NOTE** | The whole given configuration which comprises of `itf-maven-plugin` and the configuration for `maven-failsafe-plugin` should be replaced by separate maven plugin later to make the usage more convenient. |

## 4.2. Developer's Point of View

You are a potential contributor or just interested in how the code of this extension is working/looks like or you are a user who wants to test with the bleeding edge of this extension.

The requirements for building this extension is:

- JDK8+ (need to reconsider)
- Apache Maven 3.6.0+

You have to clone the git repository and you can build the extension simply via: `mvn clean install`. The `install` is needed to install the created artifacts into your local repository for reuse.

# 5. Structuring Integration Tests

## 5.1. 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 to 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 a test case implies 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, what 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.

## 5.2. 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 its own local repository (aka local cache) `.m2/repository`. Furthermore you see that the project is being 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 has been 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 wrote into `mvn-arguments.log`.

## 5.3. Several Test Cases

If we like to define some 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 com.soebes.itf.extension.assertj.MavenITAssertions.assertThat;

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

@MavenJupiterExtension
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 is 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.

# 6. Goals, Profiles, Properties and Command Line Options

## 6.1. 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*

```
@MavenJupiterExtension
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? Very brief the default set of goals which will be executed if not defined otherwise is `package` (See for a more detailed explanation). 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 are `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 add the `@MavenGoal` annotation `@MavenGoal("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` (See for a more detailed explanation).

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

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

The used `@MavenGoal` will overwrite any goal which is predefined. The given goals in `@MavenGoal` support replacement of placeholders which currently supports the following:

- ${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:

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

The equivalent on command line would be:

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

## 6.2. Profiles

In some cases you need to activate one or more Maven profiles for test purposes etc. This can be achieved by using `@MavenProfile` which looks like this:

```
@MavenJupiterExtension
class BasicIT {

  @MavenTest
  @MavenProfile("run-its")
  void first(MavenExecutionResult result) {
  }
```

If you like to activate multiple Maven profiles you can simply repeat the annotation like this:

```
@MavenJupiterExtension
class BasicIT {

  @MavenTest
  @MavenProfile("run-its")
  @MavenProfile("run-e2e")
  void first(MavenExecutionResult result) {
  }
```

An alternative is to use the array like definition like this:

```
@MavenJupiterExtension
class BasicIT {

  @MavenTest
  @MavenProfile({"run-its","run-e2e"})
```

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

The profiles can also be activated for a bunch of test cases that you don't need to add the annotation on each test case. So you can define the Maven profile on the class level instead.

```
@MavenJupiterExtension
@MavenProfile({"run-its","run-e2e"})
class BasicIT {

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

If you want to explicitly deactivate a profile this can be achieved like this:

```
@MavenJupiterExtension
class BasicIT {

  @MavenTest
  @MavenProfile("+run-its")
  void first(MavenExecutionResult result) {
  }
```

The `` ` `` prefix is responsible for deactivating a given profile during the build. We strongly recommend to use `` ` `` sign (instead of `!`) to prevent issues on different operating systems.

So finally let use show how the inheritance behaviour of the `MavenProfile` annotation is defined.

```
@MavenJupiterExtension
@MavenProfile({"run-its"})
class BasicIT {

  @MavenTest
  @MavenProfile("run-its-delta")
  void first(MavenExecutionResult result) {
    ..
  }

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

  @Nested
  @MavenProfile("nested-class")
  class NestedClass {
```

```
    @MavenTest
    @MavenProfile("nested-first")
    void nested_first(MavenExecutionResult result) {
      ..
    }
    @MavenTest
    void nested_second(MavenExecutionResult result) {
      ..
    }
  }
}
```

So based on the given example the method `first` will be executed with the activation of the profiles `run-its` and `run-its-delta`. The method `second` will be execute with the activated profile `run-its`. The method `nested_first` will be executed with the activated profiles `run-its`, `nested-class`and `nested-first`. Finally the method `nested_second` will be executed with the activated profiles `run-its` and `nested-class`.

## 6.3. System Properties

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
```

The equivalent for using such properties (which contains name and value) within an integration tests can be achieved by using the `@SystemProperty` annotation which look like this:

*CompareDependenciesIT.java*

```
package org.codehaus.mojo.versions.it;

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

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

@MavenJupiterExtension
class CompareDependenciesIT
{
    @MavenGoal("versions:set")
    @SystemProperty(value = "generateBackups", content = "false")
    @SystemProperty(value = "newVersion", content = "2.0")
    @MavenTest
    void it_compare_dependencies_001( MavenExecutionResult result )
    {
       ...
```

```
    }
}
```

In other cases it's only necessary to define the name of the property which you do on the command like this:

```
mvn clean verify -DskipTests
```

The same setup can be achieved via:

*CompareDependenciesIT.java*

```
package org.codehaus.mojo.versions.it;

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

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

@MavenJupiterExtension
class RunningIT
{
    @MavenGoal("clean")
    @MavenGoal("verify")
    @SystemProperty("skipTests")
    @MavenTest
    void running( MavenExecutionResult result )
    {
       ...
    }
}
```

During the designing of your integration tests you might realize you need the property setup for a larger number of integration test cases. You can define a property or even several properties on a class level like this:

*CompareDependenciesIT.java*

```
package org.codehaus.mojo.versions.it;

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

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

@MavenJupiterExtension
@SystemProperty("skipTests")
```

```
class RunningIT
{

    @MavenTest
    void running_first( MavenExecutionResult result )
    {
       ...
    }

    @MavenTest
    void running_second( MavenExecutionResult result )
    {
       ...
    }

    @MavenTest
    void running_thrid( MavenExecutionResult result )
    {
       ...
    }
}
```

This prevents the repetition of the property on each test case method.

Another option is to define the `SystemProperty` annotation on different levels. For example class level, method level or nested levels. Let us make deep dive into the following example:

```
..
@MavenJupiterExtension
@SystemProperty("skipTests")
class RunningIT
{

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

    @MavenTest
    @SystemProperty("second")
    void second( MavenExecutionResult result )
    {
       ...
    }

    @Nested
    @SystemProperty("NestedLevel")
    class NestedLevel {
      @MavenTest
```

```
      void nested_first( MavenExecutionResult result )
      {
         ...
      }

      @MavenTest
      @SystemProperty("NestedLevelSecond")
      void nested_second( MavenExecutionResult result )
      {
         ...
      }

   }
}
```

The method `first` will be executed with the defined system property `skipTests` given by the class level `RunningIT`. The method `second` will be executed with the defined system properties `skipTests` and also with the `second` property. The method `nested_first` will be executed with the following system properties `skipTests` and `NestedLevel` while the method `nested_second` will be executed with the following system properties `skipTest`, `NestedLevel` and `NestedLevelSecond`.

## 6.4. 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 `@MavenOption` annotation. There is a convenience class `MavenCLIOptions` available, which contains all existing command line options. You are not forced to use the `MavenCLIOptions` class.

```
@MavenGoal("versions:set")
@SystemProperty(value = "newVersion", content = "2.0")
@MavenOption(MavenCLIOptions.NON_RECURSIVE)
@MavenOption("--offline")
@MavenTest
void first( MavenExecutionResult result )
{
    assertThat( result ).isSuccessful();
}
```

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

```
@MavenGoal("versions:set")
@SystemProperty(value = "newVersion", content = "2.0")
@MavenOption({"-N","--offline"})
@MavenTest
void first( MavenExecutionResult result )
{
    assertThat( result ).isSuccessful();
```

```
}
```

## 6.5. Default Goals,Options

### 6.5.1. Goals

Basically to run a usual integration test it needs to be defined by which goals and options such test has to run. Let us take a look at a simple example:

```
@MavenJupiterExtension
class BasicIT {

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

The execution of the test will run with the goal `package` which is defined by default if not defined otherwise. There are defined some useful default command line options like `--errors`, `-V` and `--batch-mode` also by default if not defined otherwise. So let us take a look what otherwise means.

You can replace the default goal `package` by simply defining a different goal on your test class like this (you can also put that annotation on the method as well):

```
@MavenJupiterExtension
@MavenGoal("verify")
class BasicIT {

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

This means all consecutive tests will run with the goal `verify` instead of `package`. It is also possible to define several goals like this:

```
@MavenJupiterExtension
@MavenGoal("clean")
@MavenGoal("verify")
class BasicIT {

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

If you don't like to repeat that on each test class you can define a meta annotation like this:

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RUNTIME)
@Inherited
@MavenGoal({"clean", "verify"})
public @interface GoalsCleanVerify {
}
```

This meta annotation can now be used instead. This makes it easy to change the goal for all of your tests from a single location.

```
@MavenJupiterExtension
@GoalsCleanVerify
class BasicIT {

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

Now let us summarize the information for goals. As long as no explicit `@MavenGoal` is defined the default will be used which is `package`.

Furthermore we can define goal annotation `MavenGoal` on different levels which means class level, method level, nested classes etc. We will look at the following example:

```
@MavenJupiterExtension
@MavenGoal("clean")
class BasicIT {

  @MavenTest
  void first(MavenExecutionResult result) {
  }

  @MavenTest
  @MavenGoal("install")
  void second(MavenExecutionResult result) {
  }

  @Nested
  @MavenGoal("verify")
  class NestedLevel {
    @MavenTest
    void nested_first(MavenExecutionResult result) {
    }
    @MavenTest
    @MavenGoal("site:stage")
```

```
    void nested_second(MavenExecutionResult result) {
    }

  }
}
```

How would the above test cases executed related to the defined goal annotation? The method `first` will be executed with the goal `clean`. The method `second` will be executed with the goals `clean` and `install`. The method `nested_first` in the nested class `NestedLevel` will be executed with the goals `clean` and `verify` and finally the method `nested_second` will be exectuted with the goals `clean`, `verify` and `site:stage`.

### 6.5.2. Options

Now let us going back to the example:

```
@MavenJupiterExtension
class BasicIT {

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

This example will run the test with the following command line options:

- `-V`
- `--batch-mode`
- `--errors`

So if you like to change the command line options which are used to run an integration test you can simply define different options via the appropriate annotations:

```
@MavenJupiterExtension
@MavenOption(MavenCLIOptions.VERBOSE)
class BasicIT {

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

Using the option like this means (like for goals as mentioned before) all consecutive tests will now run with the given option which in this case is only a single command line option `--verbose` and nothing else. You can add supplemental command line options just by adding supplemental annotations to your tests like this:

```
@MavenJupiterExtension
@MavenOption(MavenCLIOptions.ERRORS)
@MavenOption(MavenCLIOptions.VERBOSE)
class BasicIT {

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

The MavenOption annotation can also be used to create a meta annotation like this:

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RUNTIME)
@Inherited
@MavenOption(MavenCLIOptions.VERBOSE)
@MavenOption(MavenCLIOptions.ERRORS)
@MavenOption(MavenCLIOptions.FAIL_AT_END)
public @interface OptionDebugErrorFailAtEnd {
}
```

The newly defined meta annotation can simply being used like this:

```
@MavenJupiterExtension
@OptionDebugErrorFailAtEnd
class BasicIT {

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

Now let us summarize the information for options. As long as no explicit `@MavenOption` is defined the default will be used which are the following:

- `-V`
- `--batch-mode`
- `--errors`

What will happen if you have a test class like this with nested class and the appropriate annotations:

```
@MavenJupiterExtension
@MavenOption(MavenCLIOptions.VERBOSE)
class BasicIT {
```

```
  @MavenTest
  @MavenOption(MavenCLIOptions.FAIL_AT_END)
  void first(MavenExecutionResult result) {
     ...
  }
  @MavenTest
  void second(MavenExecutionResult result) {
     ...
  }

  @MavenOption(MavenCLIOptions.FAIL_FAST)
  class NestedClass {
   @MavenTest
   void nested_first(MavenExecutionResult result) {
     ...
   }
   @MavenTest
   @MavenOption(MavenCLIOptions.FAIL_NEVER)
   void nested_second(MavenExecutionResult result) {
     ...
   }

  }
}
```

Let us analyze that. The method `second` will be executed with the option `--verbose` while the method `first` will be executed with two options `--verbose` and `--fail-at-end`. Now the interesting part. The method `nested_first` will be executed with the options `--verbose` plus the `--fail-fast` and the method `nested_second` will be executed with `--verbose`, `--fail-fast` and `--fail-never`. Just ignore that the combination of those options does not really makes sense. It's only to show the behaviour over different inheritance levels. In other words the behaviour of the `MavenOption` annotation is additive (except for the root level!).

### 6.5.3. Combined Annotation

Sometimes you would like to combine things from options and goals in a more convenient way. That is possible by using a custom annotation like this one:

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RUNTIME)
@Inherited
@MavenJupiterExtension
@MavenOption(MavenCLIOptions.VERBOSE)
@MavenOption(MavenCLIOptions.ERRORS)
@MavenOption(MavenCLIOptions.FAIL_AT_END)
@MavenGoal("clean")
@MavenGoal("verify")
public @interface MavenJupiterExtensionWithDefaults {
```

```
}
```

The given annotation include the usage of `@MavenJupiterExtension` which handles the loading of the extension part. By using the above defined annotation your test could look like this:

```
@MavenJupiterExtensionWithDefaults
class BasicIT {

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

So this makes it very easy to define your own default set of goals, options or maybe more. You can of course add properties etc. if you need.

# 7. Scenarios

## 7.1. 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*

```
@MavenJupiterExtension
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
```

## 7.2. 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. This can be achieved easily via the `@MavenRepository` annotation.[1] The usage looks like the following:

*MavenIntegrationExampleNestedGlobalRepoIT.java*

```
package org.it;

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

@MavenJupiterExtension
@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*

```
@MavenJupiterExtension
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 its 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
```

## 7.3. 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 them 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 indicates 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.

*ProjectIT.java*

```
@MavenJupiterExtension
@MavenPredefinedRepository
class ProjectIT {

  @MavenOption(MavenCLIOptions.VERBOSE)
  @MavenTest
  void project_001(MavenExecutionResult result) {
    assertThat(result).isSuccessful();
  }

  @MavenOption(MavenCLIOptions.VERBOSE)
  @MavenTest
  void project_002(MavenExecutionResult result) {
    assertThat(result).isSuccessful();
  }

}
```

In your test project definition you need to create the appropriate directory structure including the needed contents, 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:

*ProjectLevelIT.java*

```
@MavenJupiterExtension
class ProjectIT {

  @MavenOption(MavenCLIOptions.VERBOSE)
  @MavenTest
  @MavenPredefinedRepository
  void project_001(MavenExecutionResult result) {
    assertThat(result).isSuccessful();
  }

  @MavenOption(MavenCLIOptions.VERBOSE)
  @MavenTest
  @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
```

## 7.4. 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 having a single project and executing several maven execution on that project in the end. Such a use case looks like this:

*SetIT.java*

```
@MavenJupiterExtension
class SetIT
{
    private static final String VERSIONS_PLUGIN_SET =
      "${project.groupId}:${project.artifactId}:${project.version}:set";

    @Nested
    @MavenProject
    @TestMethodOrder( OrderAnnotation.class )
    @MavenOption(MavenCLIOptions.NON_RECURSIVE)
    @MavenGoal(VERSIONS_PLUGIN_SET)
    class set_001
    {

        @SystemProperty(value = "newVersion", content="2.0")
        @MavenTest
        @Order(10)
        void first( MavenExecutionResult result )
        {
            assertThat( result ).isSuccessful();
        }

        @SystemProperty(value = "newVersion", content="2.0")
        @SystemProperty(value = "groupId", content="*")
        @SystemProperty(value = "artifactId", content="*")
        @SystemProperty(value = "oldVersion", content="*")
        @MavenTest
        @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 `MavenJupiterExtension` 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")`.

# 8. Test Case Execution

## 8.1. 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.MavenJupiterExtension;
import com.soebes.itf.jupiter.extension.MavenTest;
import com.soebes.itf.jupiter.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();
  }

}
```

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.MavenJupiterExtension;
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;

@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();
  }
}
```

# 9. Assertions

## 9.1. Overview

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

```
@MavenJupiterExtension
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 been 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.

So next will be to check that a file in the `target` directory has been created during a test and should contain the required contents. 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 then 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 contents of the file as you like.

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

## 9.2. Assertion for Maven Log

In integration tests is necessary to check the log output of a build. This is sometimes needed because you want to check for particular output etc. which has been done by a plugin/extension etc.

In general there are currently two different outputs which can be reviewed:

- Console output (stdout)
- Error output (stderr)

These two parts are redirected into appropriate files within the integration result directory (see `the_first_test_case`). In the following example you see two output files `mvn-stdout.log` and `mvn-stderr.log`.

```
.
└─────target/
    └──── maven-its/
        └──── org/
```

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

Let us take a look into an example test case like this:

*LogoutputIT.java*

```
package com.its;

import com.soebes.itf.jupiter.extension.MavenJupiterExtension;
import com.soebes.itf.jupiter.extension.MavenCLIOptions;

@MavenJupiterExtension
class LogoutputIT {

  @MavenTest
  void basic(MavenExecutionResult result) {
    assertThat(result)
      .out()
      .warn()
      .containsExactly("Using platform encoding (UTF-8 actually) to copy filtered
resources, i.e. build is platform dependent!");
  }
}
```

Usually you will be using the injection of the `MavenExecutionResult` in your test case. This gives you the option to enhance the assertions like the following:

- `assertThat(result).out()…`.
- `assertThat(result).err()…`.

The first one `assertThat(result).out()…`. will give you access to the `stdout` of the build (which means `mvn-stdout.log`) whereas the second one give you access to the `stderr` of the build (means `mvn-stderr.log`).

In using the `..out()` it can be combined with things like:

- `.info()`
- `.warn()`
- `.debug()`

- `.error()`

Those parts will remove the appropriate prefix from each output line `[INFO]`, `[WARNING]`, `[DEBUG]` or `[ERROR]` (This includes the single space which is followed by them). From that point on you can use the usual AssertJ assertions as in the given example above `containsExactly` which implies that only a single would be allowed to have that single warning.

Furthermore if you like to get the plain log output that can be achieved by using:

- `.plain()`

That will not filter anything.

Another example of using the assertions could look like this:

```
    assertThat(result)
        .out()
        .info()
        .contains("Building Maven Integration Test :: it0033 1.0");
```

This will extract all messages with the prefix `[INFO] ` of the log and check if there is at least one line which contains the given content.

We can check for warnings like the following:

```
//    assertThat(result).out()
//        .warn()
//        .hasSize(1)
//        .allSatisfy(l -> {
//          assertThat(l).startsWith("Using platform encoding (");
//          assertThat(l).endsWith("to copy filtered resources, i.e. build is platform
dependent!");
//        });
```

You can access directly the `stdout` and/or the `stderr` of the Maven build and do things yourself if you prefer to go that way. In this case you have to add another injection to the test case (`MavenLog mavenLog` or like this `result.getMavenLog().getStdout()`).

The `stderr` output can be accessed as well like this:

```
    assertThat(result).err().plain().isEmpty();
```

The `stderr` output can be accessed as well like this:

```
    assertThat(result).err().plain().isEmpty();
```

A full-fledged example can be found `itf-examples/src/test/java/com/soebes/itf/examples/LogoutputIT.java` within the itf project.

## 9.3. Expressing Assertions

*CompareDependenciesIT.java*

```
package org.codehaus.mojo.versions.it;

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

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

@MavenJupiterExtension
class CompareDependenciesIT
{
    @MavenGoal("${project.groupId}:${project.artifactId}:${project.version}:compare-
dependencies")
    @SystemProperty(value = "remotePom", content="localhost:dummy-bom-pom:1.0")
    @SystemProperty(value = "reportOutputFile", content="target/depDiffs.txt")
    @MavenTest
    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" ) );
    }
}
```

# 10. Debugging Plugins

## 10.1. Overview

Sometimes you need to debug your code of your implemented plugin because usual tests or even integration tests are not enough. The question is how?

Generally you have to define the system property `ITF_DEBUG=true` before you run your integration

test. This will start the integration test while executing `mvnDebug` instead of `mvn` and will simply wait for a connection of a remote debugger usually from your IDE.

## 10.2. Limitations

Debugging can only being done after you have run your whole integration tests once via command line. An idea for the future might be a plugin for IDE's to support that in a more convenient way (Help is appreciated).

### 10.2.1. IDEA IntelliJ

You have to define a system property for debugging in your IDE run configuration. In the configuration you will find an entry for the VM which usually contains already `-ea` and exactly that field needs to be enhanced with the entry `-DITF_DEBUG=true`. By using this run configuration you can start the integration test just as before from within your IDE. The process will wait until you are connected.

The next step is to configure remote debugging which means to define the port `8000` as port and check if you are using JDK8 or JDK9+ in your configuration.

I recommend to start only a single integration test case via the above described run configuration, in debugging mode otherwise all processes would use the same port which would result in failure because only one can use that port.

### 10.2.2. Eclipse

Basically it should work the same.

(Feedback is welcome to give more details on that.)

### 10.2.3. Netbeans

Basically it should work the same.

(Feedback is welcome to give more details on that.)

# 11. Special Cases

## 11.1. Overview

In some situations it is needed to fail a build of a plugin/extension for testing purposes or other things which not that usual. This chapter describes such situations and shows solutions for those.

## 11.2. Failing the build

The integration testing framework contains a plugin `itf-failure-plugin` which (implied by the name) it's only purpose is to fail a build. The following shows a configuration which will produce a `[WARNING]` during the build cause the plugin needs to be configured.

```
<build>
  <plugins>
    <plugin>
      <groupId>com.soebes.itf.jupiter.extension</groupId>
      <artifactId>itf-failure-plugin</artifactId>
      <version>@project.version@</version>
      <executions>
        <execution>
          <id>basic_configration</id>
          <phase>initialize</phase>
          <goals>
            <goal>failure</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

The plugin does not bind to any life cycle phase by default which is intentionally to require a binding to a phase you like. This means also you can bind that plugin to any phase you would like to. In the previous example it is bound to `initialize`.

The following example shows a real example how the plugin has been configured correctly (only the `configuration` area). The configuration will fail the build with a `MojoExecutionException` (exeuctionException=true) and a text of the exception can given via `exception` configuration part.

```
          <configuration>
            <executionException>true</executionException>
            <exception>This is the ExecutionException.</exception>
          </configuration>
```

The final example will fail the build with a `MojoFailureException` (failureException=true) and the

text of the exception can given via `exception` configuration part as before.

```
            <configuration>
              <failureException>true</failureException>
              <exception>This is the FailureException.</exception>
            </configuration>
```

## 11.3. Project Not at Root Level

Sometimes you have a structure in your version control repository which does not fit with the assumptions which are made by the integration testing framework. As an example we will take a look onto the following example:

```
.
└─────root/
    └──── supplemental/
          └──── files
    └──── subdirectory
          ├──── src/
          └──── pom.xml
```

The `root` is the location where you check out your source. The directory `supplemental` contains other files and/or subdirectories which are not needed for testing. The `subdirectory` contains the directory with the plugin you would like to test with the framework.A usual setup described in previous chapters will not work.

The following integration test shows what you need to do to set up that kind of project layout correctly. The important part is `@MavenProjectLocation` which defines the directory where to find the `pom.xml`. In our case this is the `subdirectory` (previous layout example).

*TheIT.java*

```
package com.its;

@MavenJupiterExtension
class TheIT {

  @MavenTest
  @MavenProjectLocation("subdirectory")
  void basic(MavenExecutionResult result) {
    ...
  }
}
```

The definition can be done on method level or on class level. That is useful if you have several test cases where the `pom.xml`is being found in a subdirectory instead of being at the root level.

You can define the `@MavenProjectLocation` on the class level to define the same setup for several tests. The test cases `first` and `second` will use that. The configuration for `third` test case is using a different location as expressed with the supplemental `@MavenProjectLocation`. This is very unlikely.

*TheIT.java*

```
package com.its;

@MavenJupiterExtension
@MavenProjectLocation("subdirectory")
class TheIT {

  @MavenTest
  void first(MavenExecutionResult result) {
    ...
  }
  @MavenTest
  void second(MavenExecutionResult result) {
    ...
  }
  @MavenTest
  @MavenProjectLocation("differentsubdir")
  void third(MavenExecutionResult result) {
    ...
  }
}
```

## 11.4. Different Project Sources

Sometimes it is necessary to do the testing of your plugin in a different way than usually. For example, you would like to make the testing project setup programmatically or in other ways.

It might also happen that you want to reuse the same project setup for a number of different integration tests.

The default setup assumes that your projects which are used for the integration tests are located under `src/test/resources-its`. Those projects will be copied during the setup into the `target/maven-its` directory to execute the tests.

*TheIT.java*

```
package com.its;

@MavenJupiterExtension
class TheIT {

  @MavenTest
  void basic(MavenExecutionResult result) {
    ...
  }
```

```
}
```

### 11.4.1. Generated Project Setup

If you like to generate your project setup programmatically you can turn of the need of having to provide an appropriate project setup in `resources-its/⋯/`.

This can be achieved by defining `@MavenProjectSources(resourcesUsage=NONE)` that means in consequence you have to provide a full project setup on your own. The location where you have to create the setup can be calculated by using the parameters which are injected into the `beforeEach` method (see `MavenProjectResult`).

The location (directory) can be used within your `beforeEach` method `result.getTargetProjectDirectory()`.

There is a simple example for such setup in `MavenProjectSourcesBasicIT` for more details.

*TheIT.java*

```
package com.its;

@MavenJupiterExtension
@MavenProjectSources(resourcesUsage=NONE)
class TheIT {

  @BeforeEach
  void beforeEach(TestInfo testInfo, MavenProjectResult result) {
    //Setup a project into the appropriate location
    // on your own
  }
  @MavenTest
  void basic(MavenExecutionResult result) {
    ...
  }
}
```

### 11.4.2. Reusing Project for Several Tests

It could be useful to define a single project setup which should reused for several executions of integration tests.

In contradiction the `@MavenProject` gives the option to execute more than one integration test on the same project.

The `@MavenProjectSource(source="com/soebes/source-project")` defines the location where to find the project setup for the integration test relative to the `src/test/resources-its` directory. This location will not change based on package of the integration test class, method or alike.

Reconsider the default setup first:

*TheDefaultIT.java*

```
package com.its;

@MavenJupiterExtension
class TheIT {

  @MavenTest
  void basic(MavenExecutionResult result) {
    ...
  }
}
```

Based on the given setup the used integration project has to be located in the following directory:

```
.
└── src/
    └── test/
        └── resources-its/
            └── com/
                └── its/
                    └── TheIT/
                        └── basic/
                            ├── src/
                            └── pom.xml
```

If you define the integration test like this:

*TheIT.java*

```
package com.its;

@MavenJupiterExtension
@MavenProjectSources(source="com/soebes/source-project")
class TheIT {

  @MavenTest
  void basic(MavenExecutionResult result) {
    ...
  }
}
```

This means having the provided test project is now located in the `source-project` directory instead:

```
.
└── src/
    └── test/
        └── resources-its/
```

```
            └──── com/
                └──── soebes/
                    └──── source-project/
                            ├──── src/
                            └──── pom.xml
```

So you can reuse the same project for more than one integration test simply by creating a second integration test like this:

*TheIT.java*

```
package com.its;

@MavenJupiterExtension
@MavenProjectSources(source="com/soebes/source-project")
class TheIT {

  @MavenTest
  void basic_one(MavenExecutionResult result) {
    ...
  }

  @MavenTest
  void basic_two(MavenExecutionResult result) {
    ...
  }
}
```

This setup means to reuse the same project setup for both test cases `basice_one` and for `basic_two`.

The most important part will be visible in the `target/` directory:

```
.
└────target/
   └──── maven-its/
       └──── com/
           └──── its/
               └──── TheIT/
                   ├──── basic_one/
                   │    ├──── .m2/
                   │    ├──── project/
                   │    │    ├──── src/
                   │    │    ├──── target/
                   │    │    └──── pom.xml
                   │    ├──── mvn-stdout.log
                   │    ├──── mvn-stderr.log
                   │    └──── other logs
                   └──── basic_two/
                       ├──── .m2/
```

```
                        ├──── project/
                        │     ├──── src/
                        │     ├──── target/
                        │     └──── pom.xml
                        ├──── mvn-stdout.log
                        ├──── mvn-stderr.log
                        └──── other logs
```

The directory `project` of both cases `basic_one` and `basic_two` will contain a copy of the same setup `source-project`.

### 11.4.3. Different Setup for Every Method

You can define also different project setups for each integration test case individual if you like.

The following example uses the same location for the tests `one` and `two`.

The test methods `three` and `four` uses a different source location.

*TheIT.java*

```
package com.its;

@MavenJupiterExtension
class TheIT {

  @MavenTest
  @MavenProjectSources(source="com/soebes/base1")
  void one(MavenExecutionResult result) {
    ...
  }
  @MavenTest
  @MavenProjectSources(source="com/soebes/base1")
  void two(MavenExecutionResult result) {
    ...
  }
  @MavenTest
  @MavenProjectSources(source="com/soebes/base2")
  void three(MavenExecutionResult result) {
    ...
  }
  @MavenTest
  @MavenProjectSources(source="com/soebes/base2")
  void four(MavenExecutionResult result) {
    ...
  }
}
```

It is also possible to define the project source location on a nested class level. This would simply the previous example a bit.

*TheIT.java*

```
package com.its;

@MavenJupiterExtension
class TheIT {

  @MavenTest
  @MavenProjectSources(source="com/soebes/base1")
  void one(MavenExecutionResult result) {
    ...
  }
  @MavenTest
  @MavenProjectSources(source="com/soebes/base1")
  void two(MavenExecutionResult result) {
    ...
  }

  @Nested
  @MavenProjectSources(source="com/soebes/base-nested")
  class NestedSetup {
    @MavenTest
    void three(MavenExecutionResult result) {
      ...
    }
    @MavenTest
    void four(MavenExecutionResult result) {
      ...
    }
  }

}
```

### 11.4.4. Overwrite Location

It's also possible to define a kind of global source location for all methods within the class and overwrite that either by defining it on a nested class level or on test method level.

*TheIT.java*

```
package com.its;

@MavenJupiterExtension
@MavenProjectSources(source="com/soebes/base1")
class TheIT {

  @MavenTest
  void one(MavenExecutionResult result) {
    ...
  }
```

```
  @MavenTest
  void two(MavenExecutionResult result) {
    ...
  }

  @Nested
  @MavenProjectSources(source="com/soebes/base-nested")
  class NestedSetup {
    @MavenTest
    void three(MavenExecutionResult result) {
      ...
    }
    @MavenTest
    void four(MavenExecutionResult result) {
      ...
    }
  }

}
```

# Appendix A: MavenExecutionResult

In some situations you need access to the directories which have been created during the integration tests. That can simply being achieved by using the injection of `MavenExecutionResult` or `MavenProjectResult` depending on your needs.

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

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
```

There are existing the following information which you can use for further check etc. You can access them via:

- `result.getMavenProjectResult().getTargetBaseDirectory();` This represents the directory `the_first_test_case` in the above directory structure.
- `result.getMavenProjectResult().getTargetProjectDirectory()` This represents the directory `project` in the above directory structure.
- `result.getMavenProjectResult().getCacheDirectory()` This represents the directory `.m2` in the above directory structure.

If you only need access to the directory structure and not to the result of the build or the logging output of the Maven build you can change your integration test like this:

```
@MavenJupiterExtension
class FirstIT {
  @MavenTest
  void the_first_test_case(MavenProjectResult result) {
    ...
  }
}
```

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