**Integration Testing With Maven**

Petri Kainulainen July 16, 2012[59 comments](http://www.petrikainulainen.net/programming/maven/integration-testing-with-maven/#comments)

[Integration Testing](http://www.petrikainulainen.net/tag/integration-testing/), [Maven](http://www.petrikainulainen.net/tag/maven/), [Unit Testing](http://www.petrikainulainen.net/tag/unit-testing-tag/)

* [Facebook](http://www.facebook.com/sharer/sharer.php?u=http://www.petrikainulainen.net/programming/maven/integration-testing-with-maven/&t=Integration+Testing+With+Maven)
* [Twitter](http://www.petrikainulainen.net/programming/maven/integration-testing-with-maven/)
* [Google+](https://plus.google.com/share?url=http://www.petrikainulainen.net/programming/maven/integration-testing-with-maven/)
* [LinkedIn](http://www.linkedin.com/shareArticle?mini=true&ro=true&trk=EasySocialShareButtons&title=Integration+Testing+With+Maven&url=http://www.petrikainulainen.net/programming/maven/integration-testing-with-maven/)



Adding integration tests to a Maven build has traditionally been a bit painful. I suspect that the reason for this is that the standard directory layout has only one test directory (*src/test*).

If we want to use the standard directory layout and add integration tests to our Maven build, we have two options:

**First**, we can add our integration tests to the same directory as our unit tests. This is an awful idea because integration tests and unit tests are totally different beasts and this approach forces us to mix them. Also, if we follow this approach, running unit tests from our IDE becomes a pain in the ass. When we run tests, our IDE runs all tests found from the test directory. This means that both unit and integration tests are run. If we are “lucky”, this means that our test suite is slower than it could be, but often this means that our integration tests fail every time. Not nice, huh?

**Second**, we can add our integration tests to a new module. This is overkill because it forces us to transform our project into a multi-module project only because we want to separate our integration tests from our unit tests. Also, if our project is already a multi-module project, and we want to write integration tests for more than one module, we are screwed. Of course we can always create a separate integration test module for each tested module, but it would be less painful to shoot ourselves in the foot.

It is pretty clear that **both of these solutions suck**. This blog post describes how we can solve the problems of these solutions.

The requirements of our Maven build are:

* Integration and unit tests must have separate source directories. The *src/integration-test/java* directory must contain the source code of our integration tests and the *src/test/java* directory must contain the source code of our unit tests.
* Integration and unit tests must have different resource directories. The *src/integration-test/resources*directory must contain the resources of our integration tests and the *src/test/resources* directory must contain the resources of our unit tests.
* Only unit tests are run by default.
* It must be possible to run only integration tests.
* If an integration test fails, it must fail our build.
* The name of each integration test class must start with the prefix ‘IT’.

Let’s start by creating Maven profiles for unit and integration tests.

**Creating Maven Profiles for Unit and Integration Tests**

**First**, we need to create two Maven profiles that are described in the following:

The *dev* profile is used in the development environment, and it is the default profile of our Maven build (i.e. It is active when the active profile is not specified). This Maven profile has two goals that are described in the following:

1. It configures the name of the directory that contains the properties file which contains the configuration used in the development environment.
2. It ensures that only unit tests are run when the *dev* profile is active.

The *integration-test* profile is used for running the integration tests. The *integration-test* profile has two goals that are described in the following:

1. It configures the name of the directory that contains the properties file which contains the configuration used by our integration tests.
2. It ensures that only integration tests are run when the *integration-test* profile is active.

We can create these profiles by adding the following XML to our pom.xml file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | <profiles>      <!-- The Configuration of the development profile -->      <profile>          <id>dev</id>          <activation>              <activeByDefault>true</activeByDefault>          </activation>          <properties>              <!--                  Specifies the build.profile.id property that must be equal than the name of                  the directory that contains the profile specific configuration file.                  Because the name of the directory that contains the configuration file of the                  development profile is dev, we must set the value of the build.profile.id                  property to dev.              -->              <build.profile.id>dev</build.profile.id>              <!--                  Only unit tests are run when the development profile is active              -->              <skip.integration.tests>true</skip.integration.tests>              <skip.unit.tests>false</skip.unit.tests>          </properties>      </profile>      <!-- The Configuration of the integration-test profile -->      <profile>          <id>integration-test</id>          <properties>              <!--                  Specifies the build.profile.id property that must be equal than the name of                  the directory that contains the profile specific configuration file.                  Because the name of the directory that contains the configuration file of the                  integration-test profile is integration-test, we must set the value of the                  build.profile.id property to integration-test.              -->              <build.profile.id>integration-test</build.profile.id>              <!--                  Only integration tests are run when the integration-test profile is active              -->              <skip.integration.tests>false</skip.integration.tests>              <skip.unit.tests>true</skip.unit.tests>          </properties>      </profile>  </profiles> |

**Second**, we have to ensure that our application uses the correct configuration. When we run it by using the*dev* profile, we want that it uses the configuration that is used in the development environment. On the other hand, when we run our integration tests, we want that the tested code uses the configuration tailored for our integration tests.

We can ensure that our application uses the profile specific configuration by following these steps:

1. Configure our Maven build to load the profile specific configuration file (*config.properties*) from the configuration directory of the active Maven profile.
2. Configure the resource directory of our build (*src/main/resources*) and enable resources filtering. This ensures that the placeholders found from our resources are replaced with the actual property values that are read from the profile specific configuration file.

The relevant part of our POM file looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <filters>      <!--          Ensures that the config.properties file is always loaded from the          configuration directory of the active Maven profile.      -->      <filter>profiles/${build.profile.id}/config.properties</filter>  </filters>  <resources>      <!--          Placeholders that are found from the files located in the configured resource          directories are replaced with the property values found from the profile          specific configuration file.      -->      <resource>          <filtering>true</filtering>          <directory>src/main/resources</directory>      </resource>  </resources> |

**Additional Reading:**

* [Creating Profile Specific Configuration Files with Maven](http://www.petrikainulainen.net/programming/tips-and-tricks/creating-profile-specific-configuration-files-with-maven/)

Let’s move on and find out how we can add extra source and resource directories to our Maven build.

**Adding Extra Source and Resource Directories to Our Maven Build**

The requirements of our Maven build state that our integration and unit tests must have separate source and resource directories. Because the source code and the resources of our unit tests are found from the standard test directories, we have to create new source and resource directories for our integration tests.

We can add extra source and resource directories to our build by using the [Build Helper Maven Plugin](http://mojo.codehaus.org/build-helper-maven-plugin/). We fulfill our requirements by following these steps:

1. Add the Build Helper Maven Plugin to the *plugins* section of our POM file.
2. Configure the executions that add the new source and resource directories to our build.

**First**, we can add the Build Helper Maven Plugin to our Maven build by adding the following XML to the*plugins* section of our *pom.xml* file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <plugin>      <groupId>org.codehaus.mojo</groupId>      <artifactId>build-helper-maven-plugin</artifactId>      <version>1.9.1</version>      <executions>          <!-- Add executions here -->      </executions>  </plugin> |

**Second**, we have to configure the executions that add our new source and resource directories to our Maven build. We can do this by following these steps:

1. Create an execution that adds the new source directory (*src/integration-test/java*) to our build.
2. Create an execution that adds the new resource directory (*src/integration-test/resources*) to our build and enable resource filtering.

We can create these executions by adding the following XML to the configuration of the Build Helper Maven Plugin:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | <!-- Add a new source directory to our build -->  <execution>      <id>add-integration-test-sources</id>      <phase>generate-test-sources</phase>      <goals>          <goal>add-test-source</goal>      </goals>      <configuration>          <!-- Configures the source directory of our integration tests -->          <sources>              <source>src/integration-test/java</source>          </sources>      </configuration>  </execution>          <!-- Add a new resource directory to our build -->  <execution>  <id>add-integration-test-resources</id>  <phase>generate-test-resources</phase>  <goals>      <goal>add-test-resource</goal>  </goals>  <configuration>      <!-- Configures the resource directory of our integration tests -->      <resources>          <!--              Placeholders that are found from the files located in the configured resource              directories are replaced with the property values found from the profile              specific configuration file.          -->          <resource>              <filtering>true</filtering>              <directory>src/integration-test/resources</directory>          </resource>      </resources>  </configuration>  </execution> |

**Additional Reading:**

* [Build Helper Maven Plugin – Goals](http://mojo.codehaus.org/build-helper-maven-plugin/plugin-info.html)
* [Build Helper Maven Plugin – Usage](http://mojo.codehaus.org/build-helper-maven-plugin/usage.html)

Let’s find out how we can run our unit tests with Maven.

**Running Unit Tests**

If we think about the requirements of our Maven build, we notice that we must take the following things into account when we run our unit tests:

* Unit tests are run only when the *dev* profile is active.
* When we run our unit tests, our build must run all test methods found from the test classes whose name don’t start with the prefix ‘IT’.

We can run our unit tests by using the [Maven Surefire plugin](http://maven.apache.org/surefire/maven-surefire-plugin/). We can configure it by following this steps:

1. Configure the plugin to skip unit tests if the value of the *skip.unit.tests* property is true.
2. Configure the plugin to ignore our integration tests.

We can configure the Maven Surefire plugin by adding the following XML to the *plugins* section of our POM file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <plugin>      <groupId>org.apache.maven.plugins</groupId>      <artifactId>maven-surefire-plugin</artifactId>      <version>2.18</version>      <configuration>          <!-- Skips unit tests if the value of skip.unit.tests property is true -->          <skipTests>${skip.unit.tests}</skipTests>          <!-- Excludes integration tests when unit tests are run -->          <excludes>              <exclude>\*\*/IT\*.java</exclude>          </excludes>      </configuration>  </plugin> |

We can run our unit tests by using the command: *mvn clean test -P dev*.

**Additional Reading:**

* [Maven Surefire Plugin – Usage](http://maven.apache.org/surefire/maven-surefire-plugin/usage.html)
* [Maven Surefire Plugin – Skipping Tests](http://maven.apache.org/surefire/maven-surefire-plugin/examples/skipping-test.html)
* [Maven Surefire Plugin – Inclusions and Exclusions of Tests](http://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html)

Let’s move on and find out how we can run our integration tests with Maven.

**Running Integration Tests**

If we think about the requirements of our Maven build, we notice that we have to take the following things into account when we run our integration tests:

* Integration tests are run only when the *integration-test* profile is active.
* If an integration test fails, our build must fail as well.
* When we run our integration tests, our build must run all test methods found from the test classes whose name starts with the prefix ‘IT’.

We can run our integration tests by using the [Maven Failsafe plugin](http://maven.apache.org/surefire/maven-failsafe-plugin/). We can configure it by following these steps:

1. Create an execution that invokes the *integration-test* and *verify* goals of the Maven Failsafe plugin when we run our integration tests. The *integration-test* goal runs our integration tests in the *integration-test* phase of the Maven default lifecycle. The *verify* goal verifies the results of our integration test suite and fails the build if it finds failed integration tests.
2. Configure the execution to skip integration tests if the value of the *skip.integration.tests* property is true.

We can configure the Maven Failsafe Plugin by adding the following XML to the *plugins* section of our POM file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | <plugin>      <groupId>org.apache.maven.plugins</groupId>      <artifactId>maven-failsafe-plugin</artifactId>      <version>2.18</version>      <executions>          <!--              Invokes both the integration-test and the verify goals of the              Failsafe Maven plugin          -->          <execution>              <id>integration-tests</id>              <goals>                  <goal>integration-test</goal>                  <goal>verify</goal>              </goals>              <configuration>                  <!--                      Skips integration tests if the value of skip.integration.tests                      property is true                  -->                  <skipTests>${skip.integration.tests}</skipTests>              </configuration>          </execution>      </executions>  </plugin> |

We can run our integration tests by using the command: *mvn clean verify -P integration-test*.

**Additional Reading:**

* [Maven Failsafe Plugin – Usage](http://maven.apache.org/surefire/maven-failsafe-plugin/usage.html)
* [Maven Failsafe Plugin – Skipping Tests](http://maven.apache.org/surefire/maven-failsafe-plugin/examples/skipping-test.html)

Let’s move on and summarize what we learned from this blog post.

**Summary**

This blog post has taught us three things:

* We can add extra source or resource directories to our Maven build by using the Build Helper Maven Plugin.
* We can run our unit tests by using the Maven Surefire plugin.
* We can run our integration tests by using the Maven Failsafe plugin.

You can [get the example application of this blog post from Github](https://github.com/pkainulainen/maven-examples/tree/master/integration-testing).