Notatki - Maven - Plugins

Spis treści

Pluginy 1
Maven Compiler Plugin 1
Maven Compiler Plugin i Java 8 1
Maven Compiler Plugin i Java 9+ 2
Maven PMD Plugin 3

Pluginy

Maven jest narzędziem, w którym wykonywanie zadań odbywa się poprzez uruchamianie pluginów. Co ciekawe, w ramach zadań, które już uruchamialiśmy przy pomocy `Maven`, również stosowaliśmy pluginy, nieświadomie. Pluginy służą do automatyzacji zadań, które są związane z kompilacją, testowaniem, przygotowaniem paczek wdrożeniowych, czy samym wdrożeniem aplikacji. Wspomniałem, że już korzystaliśmy z pluginów. Wynika to z podejścia **convention over configuration**, co znaczy, że nawet jeżeli nie konfigurujesz tego narzędzia samodzielnie, to zgodnie z konwencją jest ono w stanie dokonać pewnych zadań **out of the box**. Przykładem może być kompilacja kodu. `Maven` jest w stanie kompilować kod przy wykorzystaniu pluginu `maven-compiler-plugin`, którego używaliśmy nieświadomie. To teraz już mamy świadomość ☺. Jego konwencja zakłada, że źródła plików `.java` znajdują się w katalogu `src/main/java`, natomiast pliki `.properties` znajdują się w katalogu `src/main/resources`. Niedługo powiemy sobie o uruchamianiu testów naszego kodu, w tym celu również stosowane są pluginy. Konkretnie po to aby `Maven` był w stanie w trakcie budowania aplikacji uruchomić wszystkie testy automatycznie.

W kolejnych przykładach pokażemy stosowanie pluginu aby przygotować naszą aplikację jako plik `.jar`.

Pod tym linkiem umieszczam 10 najpopularniejszych pluginów mavenowych.

Maven Compiler Plugin

Domyślnie wersja Javy, która jest używana do kompilacji to `1.5`. Stąd jeżeli nie ustawimy wartości `source` i `target` to dostaniemy np. błąd mówiący:

```
[ERROR] Source option 5 is no longer supported. Use 7 or later.
[ERROR] Target option 5 is no longer supported. Use 7 or later.
```

Co jest o tyle ciekawe, że Java 5 została wydana w roku 2004. Stąd musieliśmy podać wartości `source` i `target`, aby wymusić na `Mavenie` odpowiednią wersję Javy.

Maven Compiler Plugin i Java 8

Pokazywaliśmy przykład z ustawieniem wartości `source` i `target` dla Javy 8 w tagu `properties`. Możemy

również dodać te ustawienia bezpośrednio konfiguracji pluginu:

```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    ...
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.6.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    ...
</project>
```

Maven Compiler Plugin i Java 9+

Jeżeli natomiast zależy nam aby określona w pluginie wersja Javy była równa `9` lub więcej należy ustawić konfigurację pluginu w nieco inny sposób. Należy wtedy określić pole `release`. Jednocześnie należy też zmienić wersję pluginu z 3.6.1 na 3.8.0.

Pole `release` służy do tego żeby zastąpić pola `source` i `target`. Żeby nie schodzić zbyt mocno w szczegóły, w większości przypadków będziemy stosować pole `release`. Oczywiście możemy nadal stosować pola `source` i `target` tak jak zostało to pokazane wcześniej. Stosowanie pól `source` i `target` może być przydatne wtedy jeżeli będziemy chcieli kompilować kod z inną wersją Javy a uruchamiać z inną. W większości przypadków będziemy natomiast stosować pole `release`.

Przykład poniżej:

```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    ...
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.0</version>
                <configuration>
                    <release>11</release>
                </configuration>
```

```
            </plugin>
        </plugins>
    </build>
    ...
</project>
```

Natomiast jeżeli chcielibyśmy zapisać to w polu `properties`, wyglądałoby to w ten sposób:

```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    ...
    <properties>
        <maven.compiler.release>17</maven.compiler.release>
    </properties>
    ...
</project>
```

Maven PMD Plugin

Chciałem tutaj pokazać dosyć prosty, ale często spotykany na projektach **plugin**. Służy on do automatycznego weryfikowania czy nasz kod spełnia określone założenia. Innymi słowy, **plugin** ten sprawdza jakość naszego kodu na podstawie zdefiniowanych reguł. Mamy też domyślny zestaw reguł.

Oprócz Maven PMD Plugin w praktyce można też spotkać Maven Checkstyle Plugin.

Przykładowa konfiguracja `pom.xml` z wykorzystaniem pluginu PMD:

```
<project
        xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>pl.zajavka</groupId>
    <artifactId>java-maven-examples</artifactId>
    <version>1.0.0</version>

    <properties>
        <maven.compiler.release>17</maven.compiler.release>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-pmd-plugin</artifactId>
                <version>3.10.0</version>
                <configuration>
                    <printFailingErrors>true</printFailingErrors>
                </configuration>
```

```
                <executions>
                    <execution>
                        <phase>validate</phase>
                        <goals>
                            <goal>check</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Zwróć uwagę na pole `printFailingErrors`, które musimy ustawić na `true`, aby móc zobaczyć na ekranie co jest powodem powstania błędu **PMD**. Kolejny interesujący fragment to `executions`. Dzięki takiemu zapisowi mówimy pluginowi **PMD** aby uruchomił się w fazie budowania `validate`. Uruchomiony ma natomiast zostać `goal check` pluginu **PMD**.

Żeby to teraz sprawdzić zmieńmy kod klasy `MavenCompilingJsoupExamplesRunner` na przykład poniżej. Zwróć uwagę na nieużywany import `import java.util.List;`.

```
package pl.zajavka;

import java.util.List;

public class MavenCompilingJsoupExamplesRunner {

    public static void main(String[] args) throws IOException {
        System.out.println("Hello!");
    }
}
```

Po wykonaniu przykładowo komendy `mvn compile`, na ekranie wydrukuje się informacja pokazana poniżej, a `mvn compile` zakończy się statusem **BUILD FAILURE**, a nie **BUILD SUCCESS**.

```
PMD Failure: pl.zajavka.MavenCompilingJsoupExamplesRunner:3
    Rule:UnusedImports Priority:4 Avoid unused imports such as 'java.util.List'.
```