Notatki - Maven - POM

Spis treści

POM

W końcu jakiś fragment kodu! (nie wchodźmy tylko proszę w dyskusję, czy konfiguracja w pliku `.xml` jest kodem, czy nie 😄)

Plik `pom.xml` opisuje zasoby jakie maja być budowane gdy uruchomimy polecenie `Maven`. Oznacza to podanie m.in. informacji jakie zależności chcemy dodać do naszego projektu, pluginy, ewentualne profile itd. Możemy też dodawać pluginy, które pozwolą nam wykorzystywać dodatkowe `goals`.

W przypadku narzędzia `Maven` stosowane jest podejście **convention over configuration**. Jeżeli trzymamy się konwencji, czyli stosujemy się do zasad/schematu, które są narzucane przez narzędzie, nie musimy dodawać dodatkowej konfiguracji wskazującej np. lokalizacje konkretnych plików lub folderów, gdyż konwencja mówi, że konkretny plik lub rodzaj pliku powinien znajdować się w podanej ścieżce. Możemy również się tej konwencji nie trzymać i konfigurować wszystko ręcznie. W praktyce natomiast **convention over configuration** jest o tyle wygodne, że jeżeli daną konwencję znamy i się jej trzymamy to ograniczamy ilość zapisanych linijek konfiguracji opierając się na domyślnych zachowaniach narzędzia. Trzymamy się też wtedy pewnych standardów, które są również popularne i znane przez innych developerów.

Struktura folderów

Wspomniałem wcześniej, że plik `pom.xml` powinien zostać umieszczony w głównym katalogu projektu `root directory`. `Maven` określa również standardową strukturę projektu. Poniżej rozrysowuję najczęściej stosowaną strukturę projektu:

```
zajavka_project
  - pom.xml

  - .mvn
    - jvm.config

  - src
    - main
      - java
```

```
      - resources
    - test
      - java
      - resources

  - target
```

Ta sama rozpiska z komentarzami:

```
zajavka_project ①
  - pom.xml ②

  - .mvn ③
    - jvm.config ④

  - src ⑤
    - main ⑥
      - java ⑦
      - resources ⑧
    - test ⑨
      - java ⑩
      - resources ⑪

  - target ⑫
```

① **zajavka_project** jest głównym katalogiem projektu, katalogiem `root`. Oczywiście możemy nazwać nasz projekt jak chcemy.

② **pom.xml** - lokalizacja pliku `pom.xml`, mówiliśmy, że plik ten ma być umieszczony w głównym katalogu projektu.

③ **.mvn** - (zwróć uwagę na kropkę!) katalog, w którym możemy określić pliki konfiguracyjne dla `Maven`. Są one wykorzystywane w momencie gdy uruchomimy jakieś zadanie, które `Maven` ma dla nas wykonać. Przykładowo możemy w tych parametrach ograniczyć ilość pamięci RAM dostępnej dla `Maven` i do tego służyłby nam plik `jvm.config`.

④ **jvm.config** - możemy tu określić przykładowe ustawienia z których ma korzystać `Maven` podczas swojej pracy.

⑤ **src** - główny katalog, w którym umieszczamy pliki naszego projektu.

⑥ **main** - katalog, w którym umieszczamy pliki projektu, ale nie umieszczamy tutaj plików dotyczących testów automatycznych.

⑦ **java** - katalog, w którym umieszczamy kod źródłowy naszej aplikacji, czyli pliki `.java`.

⑧ **resources** - katalog, w którym umieszczamy np. pliki z "propertasami", które mają być dostępne dla naszej aplikacji. Pamiętasz **resource bundle**? Pliki tego typu umieścilibyśmy tutaj.

⑨ **test** - nie omawialiśmy jeszcze tej tematyki, ale w praktyce oprócz kodu źródłowego pisze się również testy automatyczne. Będziemy o tym jeszcze rozmawiać, natomiast w tym katalogu umieszczamy pliki, które dotyczą testów naszego projektu.

⑩ **java** - katalog, w którym umieszczamy kod źródłowy **testów** naszej aplikacji, czyli pliki `.java`.

⑪ **resources** - katalog, w którym umieszczamy np. pliki z "propertasami", które mają być dostępne z poziomu **testów** naszej aplikacji. Pliki typu **resource bundle** umieścilibyśmy tutaj, ale dotyczące **testów**.

⑫ **target** - w tym katalogu `Maven` umieści pliki, które będą wynikiem zbudowanego projektu. W tym folderze `Maven` będzie również umieszczał pliki tymczasowe, które są potrzebne w trakcie budowania projektu.

Jak wygląda plik pom.xml

Skoro wiemy już, w jaki sposób wygląda konwencja struktury projektu przy korzystaniu z narzędzia `Maven`, możemy teraz przejść do samego pliku `pom.xml`.

Części, które zostaną omówione poniżej nie są wszystkimi możliwymi, jeżeli potrzebujesz informacji, które nie są tutaj podane, odsyłam do dokumentacji.

Poniżej umieszczam najmniejszy możliwy plik `pom.xml`. Najmniejszy możliwy w kontekście jakiegokolwiek sensu zapisu. Nie bawimy się tutaj w to, że można pozbyć się jeszcze jakiś informacji 😁.

```
<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"> ①
    <modelVersion>4.0.0</modelVersion> ②

    <groupId>pl.zajavka</groupId> ③
    <artifactId>java-maven-examples</artifactId> ④
    <version>1.0.0</version> ⑤
</project>
```

① **xmlns** - jest skrótem od **XML Namespace**, które można rozumieć analogicznie jak typ danych taki jak `int` albo `double`. Można też to rozumieć analogicznie do klasy w Javie, czyli taki schemat, którego ma się trzymać plik `.xml`. Ta część nas nie interesuje, pamiętajmy tylko, że ma wyglądać tak jak jest to tutaj napisane.

② **modelVersion** - określa używaną przez nas wersję modelu `POM`, którą stosujemy do opisu pliku `pom.xml`. Zapis `4.0.0` można rozumieć jako kompatybilność z `Maven` w wersji 2 oraz 3.

③ **groupId** - unikalny identyfikator organizacji, dla której projekt jest tworzony. Może być to również nazwa projektu, jeżeli przykładowo tworzymy projekt open-source. Najczęściej będziemy tutaj stosować nazwę, która odpowiada nazwom paczek w projekcie, czyli w naszym przypadku np. `pl.zajavka`. Albo jak pracowalibyśmy dla youtube.com, to byłoby to `com.youtube`. W praktyce nie musi to być jednak nazwa paczki. Jeżeli natomiast stosujemy w tym polu kropkę, to przy zapisie wybudowanego projektu do naszego lokalnego repozytorium (później zobaczysz przykład), kropka zostanie zastąpiona separatorem katalogu, czyli np. `/` dla Windows. Czyli *<groupId>pl.zajavka</groupId>* oznaczałoby utworzenie ścieżki *<lokalizacja_maven_repository>/pl/zajavka*.

④ **artifactId** - identyfikator nazwy projektu, który tworzymy. Jeżeli wygenerujemy plik `.jar` na podstawie powyższej konfiguracji, `artifactId` będzie częścią nazwy pliku `.jar`. W przypadku zapisu do repozytorium, `artifactId` będzie dalszą częścią ścieżki, pod którą zapisany zostanie plik `.jar`

⑤ **version** - numer wersji naszego projektu. Zmieniony przez nas numer wersji oznacza, że kod źródłowy aplikacji został zmieniony i pewnie funkcjonalności mogły zostać dodane, a pewne usunięte. Numer wersji również jest używany przy tworzeniu pliku `.jar`

Najmniejszy możliwy plik `pom.xml` wcale nie musi działać poprawnie, o czym mogliśmy się przekonać w materiałach. Jeżeli chcemy uruchamiać teraz komendy `mvn` musimy dodać do tego pliku nieco treści:

```
<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.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
    </properties>
</project>
```

Jeżeli teraz chcielibyśmy zainstalować powyższą aplikację w naszym lokalnym repozytorium przy wykorzystaniu narzędzia `Maven`, należałoby wykonać następującą komendę w katalogu, gdzie znajduje się plik `pom.xml`:

```
mvn install
```

Jeżeli narzędzie jeszcze nie jest zainstalowane, to powyższa komenda zwyczajnie nie zadziała, ale jeżeli taka operacja by się powiodła, to w katalogu `<maven_repository>` (u mnie domyślnie ścieżka `<maven_repository>` to *C:/Users/karol/.m2/repository*) moglibyśmy znaleźć następujący plik:

```
<maven_repository>/pl/zajavka/java-maven-examples/1.0.0/java-maven-examples-1.0.0.jar
```

Widać teraz, że powyżej zdefiniowane `groupId`, `artifactId` oraz `version` zostały odzwierciedlone przy instalacji zbudowanego projektu w naszym lokalnym repozytorium. Wcześniej zostało wspomniane o tym jakie przełożenie będzie miał zapis `<groupId>pl.zajavka</groupId>` na lokalizację pliku końcowego, tutaj jest to widoczne.

Jednocześnie chcę dodać, że powyższy plik `pom.xml` nie zawiera ani żadnych zależności zewnętrznych, ani pluginów, zatem jest to minimalny plik `pom.xml`, który możemy wykorzystać do budowania aplikacji.

Dziedziczenie POM

Wiemy już czym jest dziedziczenie, więc tego nie trzeba tłumaczyć ☺. Chcę wspomnieć tę tematykę pobieżnie ze względu na to, że w tym momencie nie jest to aż tak istotne. Na ten moment chciałbym się bardziej skupić na innych (częściej używanych) aspektach `Maven`.

Przykładowa konfiguracja dziedziczenia:

```
<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>

    <!-- Parent POM -->
    <parent>
        <groupId>pl.zajavka.parent</groupId>
        <artifactId>java-maven-examples-parent</artifactId>
        <version>0.0.1</version>
        <relativePath>../parent/pom.xml</relativePath>
    </parent>

    <artifactId>java-maven-examples</artifactId>

</project>
```

Stosując dziedziczenie **pom**ów, może wystąpić taka sytuacja, że na końcu nie będziemy w stanie zrozumieć jak wygląda ostateczna konfiguracja. Patrząc przez analogię, to trochę tak jak to, że nie jesteśmy w stanie zrozumieć która metoda wywoła się w wyniku zachowania polimorficznego. W takim wypadku z pomocą przychodzi komenda:

```
mvn help:effective-pom
```

Powyższa komenda zwróci nam tzw. `effective POM`, czyli wersję finalną naszego pliku `pom.xml` po zastosowaniu wszystkich dziedziczeń z rodziców.

Uruchamiamy Maven

Podstawową komendą aby uruchomić narzędzie `Maven` jest wpisanie w terminalu komendy:

```
mvn
```

Oprócz tego należy również podać nazwę konkretnego `lifecycle`, `phase` lub `goal`. Pamiętać należy, że każdy `build lifecycle` składa się z konkretnych faz (`phases`), gdzie każda faza oznacza kolejny etap w cyklu życia buildu (lepiej to brzmi po angielsku 😉). Najczęściej używane `lifecycle` to `default`, chyba, że chcemy wyczyścić zbudowany projekt, wtedy używamy `clean`.

Cykl `default` odpowiada za `phase`, które służą do kompilacji i finalnego pakowania projektu. Cykl `clean` odpowiada za usuwanie plików wygenerowanych przez `Maven` podczas uruchamiania buildów. Cykl `site` natomiast odpowiada za automatyczne generowanie dokumentacji dla naszego projektu.

Najczęściej stosowany jest cykl `default`, gdyż buduje on gotową paczkę z kodem. Poniżej w tabelce umieszczam często używane `phase` wraz z wyjaśnieniami.

Phase	Opis
validate	Sprawdza poprawność projektu zależnie od narzuconych przez nas kryteriów. Krok ten może być wykorzystany przez pluginy, które automatycznie sprawdzają czy trzymamy się reguł formatowania kodu narzuconych w projekcie. Ale trzeba to skonfigurować ☺.
compile	Kompiluje kod źródłowy projektu
test	Uruchomienie testów celem przetestowania skompilowanego kodu źródłowego. Testy nie powinny zależeć od kroku `package`, ani od wdrożenia kodu na serwer. O testach dowiemy się w przyszłości.
package	Spakowanie skompilowanego kodu do odpowiedniego formatu, np. `.jar`
verify	Uruchamia kontrole wyników testów aby sprawdzić, czy spełnione zostały narzucone przez nas kryteria jakościowe. I ponownie, o testach dowiemy się w przyszłości.
install	Instaluje pakiet w lokalnym repozytorium, który może później zostać wykorzystany przez inne projekty, które będziemy pisać na naszej maszynie. Ten krok był pokazywany wcześniej w notatce.
deploy	Kopiuje stworzoną paczkę do zdalnego repozytorium, aby podzielić się naszym projektem z innymi deweloperami, albo aby umożliwić wykorzystanie naszego kodu w innych projektach

Wspomniane wyżej fazy cyklu życia wykonywane są w sekwencji aby ukończyć całość `build lifecycle`. Oczywiście nie zostały tutaj poruszone wszystkie możliwe `phases`. Oznacza to, że jeżeli zastosujemy `default lifecycle` - `Maven` w pierwszej kolejności wykona fazę `validate`, później `compile`, następnie `test`. Wybierając komendę określamy, na którym kroku wykonywanie `phases` ma się zakończyć. Czyli jeżeli wpiszemy:

```
mvn package
```

To zostaną wykonane wszystkie kroki przed package, z package **włącznie**. Zwróć uwagę, że nigdzie nie pojawia się w komendzie słowo `default`. Oznacza to, że jeżeli chcemy wywołać `lifecycle` default, musimy podać bezpośrednio jego `phase`.

Możemy również napisać to w ten sposób:

```
mvn clean package
```

Oznacza to, że zanim wykonamy fazę `package`, `Maven` spróbuje posprzątać po sobie, czyli usunąć pliki oraz katalogi, które zostały wytworzone podczas uruchamiania poprzednich buildów.

Można również wywołać samo:

```
mvn clean
```

Tutaj należy również dodać, że jeżeli chodzi o `lifecycles`, to są one wykonywane niezależnie od siebie.

Możemy je zapisać razem (przykład `mvn clean package`), natomiast zostaną one wtedy wykonane w sekwencji, oddzielnie od siebie. Tak jakbyśmy wywołali te kroki oddzielnie od siebie.

Oprócz `lifecycle` i `phase` wspominaliśmy również o `goals`. Są to elementy, które składają się na `phase`. `goals` są uruchamiane podczas uruchamiania konkretnych `phase`. Może jednak wystąpić taka sytuacja, że `goal` nie będzie związane z żadnym `phase` i wtedy należy je uruchamiać ręcznie (są to raczej specyficzne przypadki).

Przykładowo, aby uruchomić `goal` bezpośrednio, można to zrobić w sposób podany poniżej:

```
mvn dependency:copy-dependencies
```