Notatki - Maven - Dependencies

Spis treści

Zależności (Dependencies)

Dochodzimy nareszcie do tematu, który pozwoli nam korzystać z zewnętrznych bibliotek w prosty, szybki i przenośny sposób. Dlaczego piszę, że w przenośny? Mając konfigurację zależności zewnętrznych w `pom.xml` możemy pobrać gotowy kod projektu i nie musimy bawić się w ręczne dodawanie bibliotek. Wszystko jest zapisane w pliku `pom.xml` i `Maven` za nas dociągnie zależności, które są nam potrzebne. Zależności, czyli zewnętrzne biblioteki.

W praktyce jeżeli chcemy korzystać z jakiejś biblioteki, oznacza to, że musi ona się znaleźć na `classpath` na etapie kompilacji kodu, a następnie na etapie uruchomienia programu.

W praktyce mogą wystąpić takie sytuacje, że nasze biblioteki wymagają pewnych zależności na etapie uruchomienia programu, ale nie na etapie kompilacji, ale na razie nie będziemy wchodzić w tę tematykę.

`Classpath` to parametr, który określa gdzie Java może znaleźć (na etapie kompilacji lub w trakcie działania) określone przez użytkownika klasy i pakiety. Przypomnij sobie, po co są importy i paczki. Nawet jeżeli stosujemy mechanizm importów i paczek, to Java musi jeszcze wiedzieć, gdzie ma znaleźć kod do uruchomienia, gdzie są te pliki z kodem. Byłoby bardzo niepraktyczne, gdyby Java musiała szukać wszystkich skompilowanych plików na całym naszym dysku. W tym celu stosowany jest parametr, który wskazuje, gdzie Java ma szukać klas i pakietów dostarczonych przez użytkownika. Trzeba tutaj natomiast wspomnieć, że jeżeli korzystamy z zewnętrznych bibliotek w naszym kodzie źródłowym (dostarczonych np. poprzez pliki `.jar`), to miejsce ich położenia musi być wskazane kompilatorowi na etapie kompilacji (bo przecież możemy potrzebować tego kodu źródłowego aby poprawnie skompilować program). Następnie, przy uruchomieniu programu **VM** (Virtual Machine) potrzebuje znać miejsce położenia tych bibliotek, aby móc je wykorzystać na etapie działania programu. Jeżeli skompilujemy kod wskazuje położenie jakiejś biblioteki, a następnie nie wskażemy gdzie ona się znajduje w trakcie działania tego programu dostaniemy error `NoClassDefFoundError` wyrzucony przez JVM.

Może pojawić się teraz pytanie, skoro o tym wszystkim piszę, dlaczego wcześniej nie zostało to wspomniane. Przecież na etapie JDBC dodawaliśmy **zewnętrzny driver** do PostgreSQL.

Dodaliśmy lokalizację tego drivera w ustawieniach projektu w Intellij, dzięki temu Intellij za nas wskazał położenie tej biblioteki. Jeżeli od tego momentu na Twoim komputerze nie zostały zmienione te ustawienia, spróbuj zerknąć na całą komendę, która jest wykonywana w momencie gdy uruchamiasz jakikolwiek program (uruchom dowolną metodę main i zobacz co drukuje się na samej górze konsoli). U mnie wygląda mniej więcej tak:

```
C:\...java.exe
    ... jakieś parametry
    -classpath
        C:\Users\karol\zajavka\out\production\examples;
        C:\Users\karol\zajavka\examples\src\pl\zajavka\java\jdbc\postgresql-42.2.23.jar
    zajavka.loops.examples.Example1
```

Oczywiście sformatowałem tę linijkę w przystępny sposób, w Intellij jest to jedna linijka tekstu. Co tutaj widzimy?

- **C:\...java.exe** - oznacza lokalizację pliku `.exe`, który faktycznie uruchomi mój program.
- **classpath** - czyli wskazanie gdzie VM może szukać klas i paczek, które są wykorzystane w programie. Wskazane są tutaj takie wpisy:
  - **C:\Users\karol\zajavka\out\production\examples;** - czyli miejsce gdzie odkładane są skompilowane pliki `.class`. W trakcie uruchomienia programu **JVM** wie, że tutaj może szukać klas i paczek, które są potrzebne
  - **C:\Users\karol\zajavka\examples\src\pl\zajavka\java\jdbc\postgresql-42.2.23.jar** - wskazanie gdzie jest położony driver JDBC do PostgreSQL
- **zajavka.loops.examples.Example1** - program który faktycznie uruchamiam, czyli bez tych wszystkich parametrów ten zapis wyglądałby tak jak poniżej i to jest zapis, który znamy:

```
java zajavka.loops.examples.Example1
```

Po co tak właściwie się o tym wszystkim rozpisuję? Bo jeżeli używamy `Maven` w zestawieniu z Intellij, to nie musimy się nad tym zastanawiać, jak to skonfigurować. Narzędzia robią to za nas.

W praktyce mogą też występować takie sytuacje, że pobierzemy jakąś bibliotekę i okazuje się, że wymaga ona 4 innych bibliotek do poprawnego działania. W tym również pomaga nam `Maven`. Dzięki temu narzędziu określamy w pliku `pom.xml` jakie biblioteki nas interesują, w jakich wersjach, a `Maven` pobierze je i zainstaluje w naszym lokalnym repozytorium. Jeżeli jakakolwiek z tych bibliotek wymaga dodatkowych bibliotek, również zostaną one pobrane i zainstalowane w lokalnym repozytorium (oczywiście w zależności od naszych ustawień).

W jaki sposób to zapisać? W **POM**ie poniżej dodajemy 2 zależności, bibliotekę `jsoup` oraz bibliotekę `guava`. Na ten moment nie ma to znaczenia, do czego one są. Każda dependencja jest dodana w tagu `dependencies` i każda z nich oddzielnie w tagu `dependency`. Zwróć uwagę, że opisując dependencję, podajemy jej `groupId`, `artifactId` oraz `version`.

```
<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>
```

```
    <dependencies>
        <dependency>
            <groupId>org.jsoup</groupId>
            <artifactId>jsoup</artifactId>
            <version>1.14.2</version>
        </dependency>
        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>31.1-jre</version>
        </dependency>
    </dependencies>
</project>
```

Jeżeli uruchomimy np. komendę `mvn package` to narzędzie pobierze te 2 zależności z **centralnego repozytorium maven** (Maven Central Repository) i zapisze je w naszym lokalnym repozytorium (czyli ta ścieżka z `.m2` w nazwie). Jeżeli już tam są, to `Maven` nie będzie ich pobierał.

Jeżeli pojawiło Ci się pytanie, skąd wiem co dokładnie wpisać w tagu `dependency` - kopiuję to stąd

Zależności a classpath

Wspomniałem wcześniej, że dzięki wykorzystaniu `Maven`, nie musimy ręcznie określać jakie biblioteki mają zostać wykorzystane na etapie kompilacji projektu, oraz nie musimy ręcznie określać jakie biblioteki mają zostać użyte przy uruchomieniu programu.

Zanim przejdę do wyjaśnienia czym jest `scope`, parę przykładów.

Najpierw spójrzmy na classpath

Wyobraźmy sobie, że chcemy ręcznie skompilować projekt składający się z kilku klas. Struktura plików wygląda w ten sposób:

```
<some_dir>/pl/zajavka/MavenCompilingExamplesRunner.java
<some_dir>/pl/zajavka/animal/Cat.java
<some_dir>/pl/zajavka/animal/Dog.java
```

A kolejne pliki źródłowe w ten sposób:

```
package pl.zajavka;

import pl.zajavka.animal.Cat;
import pl.zajavka.animal.Dog;

public class MavenCompilingExamplesRunner {

    public static void main(String[] args) {
        System.out.println("Running...");
        System.out.println(new Cat("Romek").getName());
        System.out.println(new Dog("Burek").getName());
    }
}
```

```
package pl.zajavka.animal;

public class Cat {

    private final String name;

    public Cat(final String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

```
package pl.zajavka.animal;

public class Dog {

    private final String name;

    public Dog(final String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Znajdując się teraz w katalogu głównym, czyli w `<some_dir>`, możemy teraz uruchomić kompilator w taki sposób:

```
javac pl/zajavka/MavenCompilingExamplesRunner.java
```

Zobacz, że dzięki temu otrzymaliśmy 3 pliki `.class`: `MavenCompilingExamplesRunner.class`, `Cat.class` i `Dog.class`, przy czym `Cat.class` i `Dog.class` są zlokalizowane w katalogu `animal`. Jeżeli teraz chcielibyśmy uruchomić ten program, należy uruchomić komendę:

```
java pl.zajavka.MavenCompilingExamplesRunner
```

Zwróć uwagę, że występuje zgodność katalogu (z którego wykonywane są komendy) z paczką, w której zdefiniowana jest klasa `MavenCompilingJsoupExamplesRunner`. Jeżeli uruchomimy samą komendę:

```
java MavenCompilingExamplesRunner
```

To dostaniemy błąd:

```
Error: Could not find or load main class MavenCompilingExamplesRunner
```

```
Caused by: java.lang.ClassNotFoundException: MavenCompilingExamplesRunner
```

Ważne jest to żeby zauważyć, że kompilator oraz VM same znalazły położenie klas `Cat` i `Dog`, nie musieliśmy podawać tego ręcznie.

Co natomiast jeżeli będziemy chcieli wykorzystać kod z jakiejś zewnętrznej biblioteki, np. `jsoup`? Napiszmy taki kod:

```
package pl.zajavka;

import org.jsoup.Jsoup;

import java.io.IOException;

public class MavenCompilingJsoupExamplesRunner {

    public static void main(String[] args) throws IOException {
        System.out.println(Jsoup.connect("https://app.zajavka.pl").get().title());
    }
}
```

Biblioteka `jsoup` służy do parsowania źródła `html` podanej strony internetowej.

Jeżeli teraz postaramy się wykonać poniższe polecenie:

```
javac pl/zajavka/MavenCompilingJsoupExamplesRunner.java
```

Dostaniemy poniższy komunikat:

```
pl\zajavka\MavenCompilingJsoupExamplesRunner.java:3: error: package org.jsoup does not exist
import org.jsoup.Jsoup;
               ^
pl\zajavka\MavenCompilingJsoupExamplesRunner.java:10: error: cannot find symbol
        System.out.println(Jsoup.connect("https://app.zajavka.pl").get().title());
                           ^
  symbol:   variable Jsoup
  location: class MavenCompilingJsoupExamplesRunner
2 errors
```

Czyli kompilator mówi nam: "Hej, nie wiem czym jest org.jsoup.Jsoup i gdzie mam to znaleźć!" Jeżeli natomiast umieścimy lokalizację biblioteki `jsoup-1.14.2.jar` na `classpath` w ten sposób:

```
javac -cp C:/Users/karol/pl/zajavka/jsoup-1.14.2.jar; pl/zajavka/MavenCompilingJsoupExamplesRunner.java
```

lub

```
javac -classpath C:/Users/karol/pl/zajavka/jsoup-1.14.2.jar;
pl/zajavka/MavenCompilingJsoupExamplesRunner.java
```

Czyli podałem absolutną lokalizację, gdzie kompilator może znaleźć bibliotekę `jsoup`. Kod skompilował się poprawnie. Jeżeli teraz spróbuję go uruchomić:

```
java pl.zajavka.MavenCompilingJsoupExamplesRunner
```

Dostanę poniższy błąd:

```
Exception in thread "main" java.lang.NoClassDefFoundError: org/jsoup/Jsoup
        at pl.zajavka.MavenCompilingJsoupExamplesRunner.main(MavenCompilingJsoupExamplesRunner.java:10)
Caused by: java.lang.ClassNotFoundException: org.jsoup.Jsoup
        at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:606)
        at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:168)
        at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:522)
        ... 1 more
```

Spróbujmy zatem w ten sposób:

```
java -cp C:/Users/karol/pl/zajavka/jsoup-1.14.2.jar; pl.zajavka.MavenCompilingJsoupExamplesRunner
```

lub

```
java -classpath C:/Users/karol/pl/zajavka/jsoup-1.14.2.jar; pl.zajavka.MavenCompilingJsoupExamplesRunner
```

Na ekranie (na moment pisania tego tekstu) wydrukuje się: `Platforma Zajavka - programowanie w Javie`.

Wniosek? Jeżeli korzystamy z zewnętrznych bibliotek, musimy ręcznie na etapie kompilacji `javac` i uruchomienia programu `java` podać lokalizację bibliotek, które są wykorzystywane w naszym kodzie.

Tylko, że my w praktyce korzystamy z Intellij, żeby było łatwiej i szybciej. Przynajmniej wiemy już co Intellij robi za nas ☺. Czyli w przypadku, gdy dodawaliśmy driver do `PostgreSQL`, zrobiliśmy to z poziomu Intellij. Czyli każda osoba, która ściąga sobie projekt do edycji musiałaby na swoim komputerze ponawiać ten proces dodawania bibliotek. Pamiętajmy, że kod w pracy cały czas żyje, ciągle są dodawane nowe biblioteki. Zatem uciążliwe by było dodawanie tych bibliotek (albo usuwanie) za każdym razem ręcznie z poziomu okna Intellij. I z pomocą przychodzi nam `Maven` i jego integracja z Intellij.