Notatki - Maven - Intro

Spis treści

Czym jest Maven?

Maven jest narzędziem, które służy do m.in. budowania. Nie jest to oczywiście jedyne narzędzie, które do tego służy. Czym jest samo budowanie projektu, wyjaśni się w trakcie.

W trakcie trwania warsztatów poruszymy podstawowe koncepcje działania tego narzędzia, nie wspomnimy natomiast o wszystkich możliwych jego mechanizmach - o tym często przekonasz się już w pracy w praktyce. Skupimy się na zrozumieniu najważniejszych koncepcji.

Obraz 1. Maven logo. Źródło: https://pl.wikipedia.org/

Czym jest budowanie?

Wspomnieliśmy, że **Maven** jest narzędziem do budowania. Na stwierdzenie `budowanie` mogą składać się poniższe czynności:

- **zarządzanie zależnościami** - dotychczas dociągaliśmy do naszego projektu bardzo mało zależności zewnętrznych. Czyli gotowych fragmentów kodu, pozwalających osiągnąć nam jakiś cel szybciej. Nie musimy wtedy takiego kodu pisać na własną rękę. Narzędzia do budowania pozwalają nam w prosty sposób zarządzać takimi zależnościami zewnętrznymi w naszym projekcie.
- **kompilacja kodu źródłowego** - poznaliśmy już tę czynność wcześniej. Można było uruchomić kompilację programu albo za pomocą komendy `javac`, albo przy wykorzystaniu Intellij i przycisku `run` (tutaj kod się kompilował i od razu uruchamiał).
- **generowanie dokumentacji** - w kodzie źródłowym możemy opisywać nasze klasy i metody przy wykorzystaniu `javadoc`, a następnie przygotować w ten sposób dokumentację projektu, która będzie podobna do np. Dokumentacji klasy String.

- **generowanie kodu źródłowego** - w praktyce kod źródłowy może być oczywiście pisany (i to jest to czego się uczymy cały czas). Może natomiast też wystąpić sytuacja, w której kod źródłowy będziemy generować automatycznie. Wtedy z pomocą przyjdą nam narzędzia do budowania i pluginy (rozszerzenia) pozwalające na generację kodu.
- **przygotowywanie paczek** - jeżeli chcielibyśmy nasz program Java wysłać komuś w postaci jednego pliku, który może zostać uruchomiony, to możemy w tym celu przygotować archiwum `.jar` lub `.zip`. Plik `.jar` (Java archive) został już pokazany w przypadku korzystania z JDBC, gdzie potrzebowaliśmy pobrać sterownik do bazy danych. Sterownik taki był dostarczony w pliku `.jar`. Pokażemy również, jak przygotować program w pliku `.jar`, który można uruchomić mając plik `.jar`.
- **instalacja programu na serwerze** - po ukończeniu programu możemy zainstalować go na serwerze (na innym komputerze dostępnym w sieci, z którego nasz program będzie mógł być uruchamiany).

Oprócz czynności wymienionych powyżej, narzędzia służące do budowania projektów, mogą wykonywać wiele więcej. Wszystkie z tych czynności mogą być oczywiście wykonywane ręcznie. Narzędzia tego typu powstają natomiast aby automatyzować procesy tego typu i eliminować możliwość popełnienia błędu przez ludzi. Z reguły najczęstszym powodem powstania błędu w projektach informatycznych jest czynnik ludzki ☺. Do tego jest szybciej, komputer klika szybciej niż człowiek.

Słowa kluczowe

Jak z każdym narzędziem, dochodzi nam kwestia terminologii, która jest nowa i którą trzeba wyjaśnić. Poniżej wyjaśniam tylko skrótowo pojęcia, w dalszych częściach materiału będziemy rozmawiać o nich bardziej szczegółowo.

POM

`POM - Project Object Model`. **POM** jest plikiem w formacie `.xml`, który zawiera opis tego jakie zasoby w naszym projekcie powinny zostać zbudowane. Wspomniałem również o zarządzaniu zależnościami. W **POM**ie dodajemy informacje o zależnościach, które są nam potrzebne w naszym projekcie. Plik `pom.xml` powinien zostać umiejscowiony "na górze" struktury katalogów naszego projektu (zobaczysz w praktyce co mam na myśli). Jednocześnie też, jeżeli wywołujemy proces budowania, musimy wskazać lokalizację pliku `pom.xml` na dysku (po to żeby narzędzie wiedziało, gdzie jest opis tego co i jak ma zbudować).

Build lifecycle

W procesie budowania projektu możemy wyróżnić pojęcia takie jak: `lifecycle`, `phase` oraz `goal`. Lifecycle składa się z `phases`, a każda `phase` składa się z `goals`. Domyślnie wbudowane mamy 3 rodzaje `lifecycle`: `default`, `clean` i `site`. (Cały czas będę starał się używać angielskich nazw, wybacz mi łamańce językowe)

- **default** - główny `lifecycle`, odpowiedzialny za utworzenie zbudowanej wersji projektu i jej ewentualne wdrożenie.
- **clean** - odpowiedzialny za wyczyszczenie plików powstałych w wyniku uruchomienia poprzedniego builda.
- **site** - pozwala na utworzenie strony internetowej z dokumentacją naszego projektu.

Głównym `lifecycle` jest `default`.

Dalej nie rozpisuje się o konkretnych `phase` oraz `goals` bo zobaczysz, że w praktyce będziesz pamiętać tylko parę z nich, a resztę trzeba będzie Googlować w indywidualnych przypadkach. Dla zainteresowanych umieszczam link Life cycles, Phases, Goals

Wdrożenie jest stwierdzeniem, pod którym kryje się fizyczne umieszczenie kodu naszej aplikacji na serwerze, na którym ma być ona uruchamiana. W pracy często spotkasz sie ze stwierdzeniem "wdrożenie na PROD" lub "wdrożenie na TEST".

Najpierw wyjaśnijmy czym są tak zwane "środowiska". Jeżeli jesteś klientem jakiegoś banku prawdopodobnie możesz dostać się do aplikacji webowej swojego banku poprzez stronę `https://www.mojwymyslonybank.pl`. Strona ta jest dostępna dla klienta, czyli Ciebie jako klienta banku. Przed wdrożeniem aplikacji (czyli faktycznym umieszczeniem napisanego kodu na serwerze, na którym taka aplikacja będzie działała) należy jeszcze taką aplikację przetestować (czyli pochodzić po stronie internetowej i poklikać czy nic się nie popsuło w wyniku dodawanych zmian w kodzie). Jeżeli aplikacja już działa i dodajemy do niej dodatkowe funkcjonalności, również należy taką aplikację przetestować. W idealnym świecie mamy do tego tak zwane środowiska DEV (deweloperskie) oraz TEST (testowe). Możemy również mieć takich środowisk więcej, jest to kwestia polityki firmy. Środowiska DEV i TEST nie są dostępne na zewnątrz firmy. Oznacza to, że istnieją takie strony jak np. `https://www.mojwymyslonybank-d.pl` lub `https://www.mojwymyslonybank-t.pl`, pod którymi znajdziemy wersję aplikacji, która nie jest jeszcze wdrożona na środowisko produkcyjne, a może zawierać nowe funkcjonalności. Strony takie powinny być dostępne tylko z sieci firmowej, czyli dopiero jak pójdziesz do biura i połączysz się tam z Internetem (albo połączysz się do sieci firmowej w inny sposób) to będziesz w stanie zobaczyć te strony u siebie w przeglądarce.

Stwierdzenia "wdrożenie na PROD" lub "wdrożenie na TEST" oznaczają umieszczenie napisanego przez Ciebie kodu na każdym z serwerów, który dostarcza nam aplikację dostępną pod adresami `mojwymyslonybank-d` lub `mojwymyslonybank-t`.

Dependencies

Każdy większy projekt posiada pewne zależności. Mówiąc zależności mam na myśli zewnętrzne biblioteki (libraries), które mogą zostać przez nas wykorzystane do wytworzenia naszego oprogramowania.

`Biblioteka` (library) jest gotowym napisanym przez kogoś innego kodem, z którego możemy korzystać. Pisząc aplikacje w pracy, nie piszemy wszystkiego sami. Korzysta się z bardzo dużej ilości gotowych rozwiązań, które pozwalają nam reużyć kod napisany przez kogoś innego. Często ktoś inny i tak zrobi to za nas lepiej (szybciej, krócej), głównie dlatego, że często jest specjalistą w temacie. My dzięki temu możemy skupić się na dostarczeniu oprogramowania, dzięki któremu firma, dla której pracujemy faktycznie zarobi pieniądze.

Zależności takie są dostarczane w postaci plików `.jar`.

Repositories

Jeżeli już wspominamy o terminie `dependencies` należy również wspomnieć o terminie `repositories`. Zależności takie, po pobraniu do nas na komputer, są przechowywane w tak zwanym `lokalnym repozytorium` (local repository) - wspomnimy o tym później. Zanim natomiast pobierzemy takie zależności do nas, musimy ich wyszukać w jakimś repozytorium, np. Maven Central.

Żebyśmy też wiedzieli o czym mówię, umieszczam link do biblioteki Lombok, z której będziemy korzystać później. Widać tutaj wersje biblioteki, z których możemy korzystać.

Oprócz `Maven Central` możemy również zdefiniować inne dostępne w internecie repozytoria.

Build Plugin

Do konfiguracji budowania projektu możemy też dołożyć pluginy. Plugin jest oprogramowaniem dokładającym do naszego cyklu budowania specyficzne akcje, które są nam potrzebne na etapie budowania naszego projektu. Akcje takie np. nie są dostępne standardowo i musimy dołożyć je na własną rękę. Przykładowo, jeżeli na etapie budowania projektu będziemy chcieli wygenerować gotowy kod, którego nie chcemy pisać sami (po co, jeżeli można go wygenerować), to do takiej akcji możemy potrzebować pluginu do generowania kodu.

Build Profile

Profile budowania są wykorzystywane jeżeli chcemy wskazać różnice między sposobami budowania naszej aplikacji w zależności od tego, na które środowisko ma zostać ona wdrożona (`DEV`, `TEST`, `PROD`). Możemy wtedy włączać lub wyłączać konkretne operacje lub ustawienia, w zależności od środowiska, które nas interesuje. Mielibyśmy wtedy np. profil `DEV`, `TEST` oraz `PROD` i każdy profil przygotowywałby zbudowany projekt na dedykowane środowisko.

W praktyce spotkasz się ze stwierdzeniem **build**. Stwierdzenie "Przygotuj builda" oznacza zbudowanie projektu. Przykładowo, plik końcowy, który może zostać umieszczony na serwerze nazywany jest `buildem`. Buildy mogą też mieć wersje, oznacza to kolejne wersje aplikacji, zbudowane i wdrożone na serwer. Czasem można też spotkać się ze stwierdzeniem `solucja`, ale w praktyce słyszałem to bardzo, bardzo rzadko.

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
```

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.

Notatki - Maven - Intellij

Spis treści

Integracja Maven z Intellij

Najpierw sam Maven, bez Intellij. Dodajmy teraz w pliku pom.xml poniższą konfigurację.

```
<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>

    <dependencies>
        <dependency>
            <groupId>org.jsoup</groupId>
            <artifactId>jsoup</artifactId>
            <version>1.14.2</version>
        </dependency>
    </dependencies>
</project>
```

Pamiętajmy jednocześnie o podejściu **Convention over configuration**.

W głównym katalogu projektu (czyli tam, gdzie jest plik pom.xml) umieszczamy folder: src/main/java i dopiero w nim umieszczamy katalog pl/zajavka. W nim umieszczamy plik MavenCompilingJsoupExamplesRunner.java. Czyli struktura wygląda teraz tak:

```
główny_katalog_projektu
  - pom.xml
  - src
    - main
```

```
      - java
        - pl
          - zajavka
            MavenCompilingJsoupExamplesRunner.java
  - target
```

Teraz (w terminalu) będąc w folderze **główny_katalog_projektu** możemy uruchomić polecenie:

```
mvn compile
```

Na ekranie wydrukowane zostanie coś podobnego do:

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< pl.zajavka:java-maven-examples >-------------------
[INFO] Building java-maven-examples 1.0.0
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ java-maven-examples ---
...
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ java-maven-examples ---
[INFO] Changes detected - recompiling the module!
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.148 s
[INFO] Finished at: 2021-09-30T13:48:40+02:00
[INFO] ------------------------------------------------------------------------
```

Jeżeli natomiast chcemy aby `Maven` wydrukował o wiele więcej informacji to napiszemy:

```
mvn compile -X
```

Informacja **BUILD SUCCESS** mówi nam o tym, że `Maven` z powodzeniem zakończył swoje działanie. Nie oznacza to natomiast, że zrobił dokładnie to co chcieliśmy bo zawsze mogliśmy popełnić gdzieś błąd.

Maven compiler plugin

Co oznacza w konfiguracji `pom.xml` taki zapis:

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

Bardzo dobrze wyjaśnia to dokumentacja

Czyli jeżeli chcemy używać funkcji, które zostały dodane w Java 17, dodajemy opcję `source` z

parametrem 17. Jeżeli chcemy, aby skompilowane klasy były kompatybilne z JVM w wersji 17, dodajemy zapis `target`. Dla innych wersji Javy upewnij się proszę z dokumentacją zanim dokonasz zmian ☺.

Jeżeli teraz zmienię zmienię zawartość pliku `.java` i dodam do niego metodę `List.of()`, której nie ma w Javie 7, oraz ustawię **source** i **target** na **1.7**:

```
package pl.zajavka;

import org.jsoup.Jsoup;

import java.io.IOException;
import java.time.LocalDateTime;
import java.util.List;

public class MavenCompilingJsoupExamplesRunner {

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

        List.of("");
    }
}
```

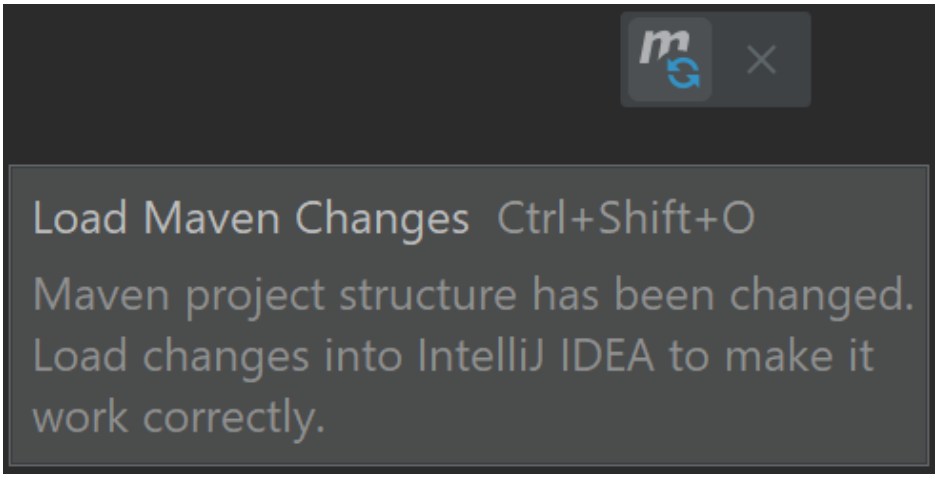
Dostanę poniższy błąd:

```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.1:compile
    (default-compile) on project java-maven-examples: Compilation failure
[ERROR] .../pl/zajavka/MavenCompilingJsoupExamplesRunner.java:[14,13]
    static interface method invocations are not supported in -source 7
[ERROR]   (use -source 8 or higher to enable static interface method invocations)
[ERROR]
[ERROR] -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException
```

Czyli widzimy już, że `Maven` pomaga nam automatycznie 'zaciągać' zależności - biblioteki zewnętrzne.

Dołóżmy do tego Intellij

Jeżeli dopiszemy `dependencies` w pliku `pom.xml` oraz mamy poprawnie skonfigurowany projekt, nie musimy nawet wywoływać z terminala komendy `mvn compile`, albo innej. W prawym górnym rogu Intellij wyświetli nam ikonkę, która oznacza możliwość odświeżenia dopisanych zależności.



Obraz 1. Load Maven Changes

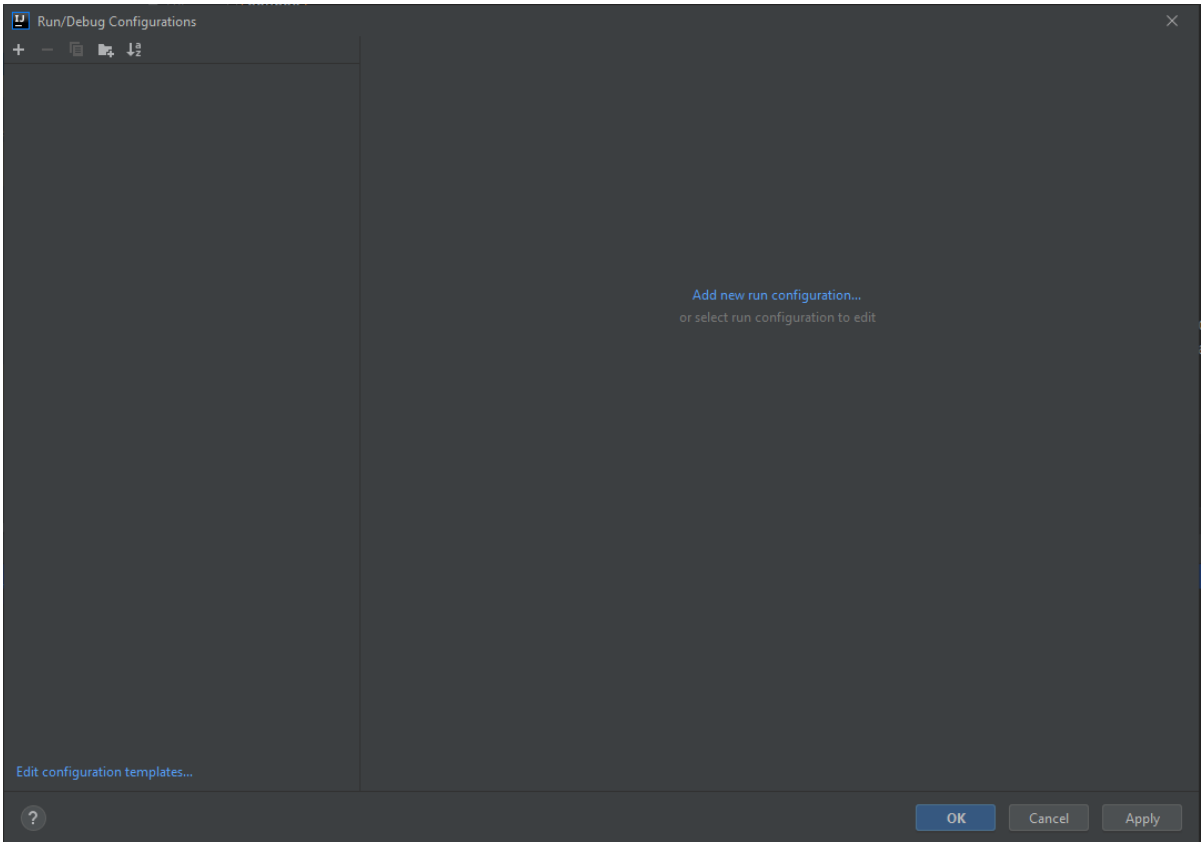
W tym momencie następuje pobranie wymaganych zależności. Jeżeli teraz chcemy uruchomić nasz projekt, wystarczy kliknąć zieloną strzałkę, którą już doskonale znamy. Intellij dodaje nam lokalizację bibliotek sam, do komendy wywołującej program.

```
C:\...\java.exe
    ...
    -classpath
        C:\Users\karol\java-pl\zajavka\examples-maven\target\classes;
        C:\Users\karol\.m2\repository\org\jsoup\jsoup\1.14.2\jsoup-1.14.2.jar
    pl.zajavka.MavenCompilingJsoupExamplesRunner
```

W ten sposób wiemy już jak działa automatyzacja związana z integracją maven i Intellij. Ale to nie wszystko.

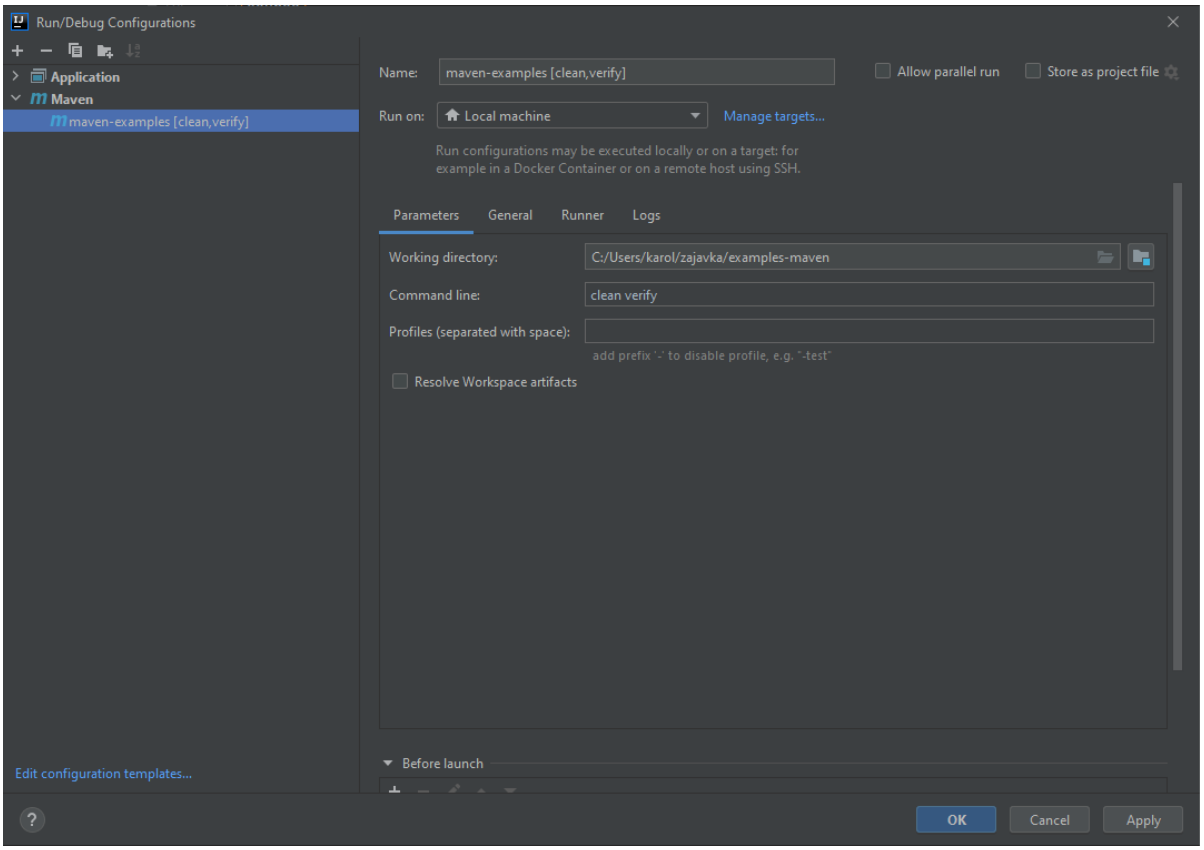
Uruchamianie poleceń Maven z Intellij

Jeżeli wciśniesz teraz dwukrotnie `Shift` i wpiszesz `edit configurations` pokaże Ci się okno, którym można definiować zadania uruchomieniowe. Dokładnie to samo robi Intellij gdy uruchamiamy nasz program z metody main. Dodawana jest wtedy konfiguracja uruchomieniowa programu. Możemy również sami zdefiniować taką konfigurację w przypadku Maven.



Obraz 2. Tworzenie konfiguracji uruchomienia Maven z Intellij

Następnie możemy wybrać '+' po lewej stronie i wybrać konfigurację `Maven`. Możemy wtedy stworzyć taką konfigurację:



Obraz 3. Własna konfiguracja Maven z Intellij

Możemy wybrać teraz konfigurację o nazwie `maven-examples [clean,verify]` i uruchomić Maven z poziomu Intellij.

To co Intellij tak na prawdę zrobi to uruchomi polecenie, które można zobaczyć jako pierwsze w oknie z konsolą, gdzie drukowany jest rezultat wywołania `Maven`.

Okno Maven po prawej stronie ekranu

Maven również dodaje nam możliwość zobaczenia drzewo `lifecycle`, `phase` i `goals` w oknie po prawej stronie ekranu. Okno to pokazuje również zdefiniowane przez nas konfiguracje. Możemy też tutaj zobaczyć biblioteki, które zostały pobrane do nas na maszynę.

Notatki - Maven - Scope

Spis treści

Czym jest scope?

Wróćmy jednak do jeszcze jednej kwestii, skoro już wiemy czym jest `classpath`. Czym jest scope? Widzieliśmy już taki zapis:

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.20</version>
    <scope>provided</scope>
</dependency>
```

Głębsze wyjaśnienie oczywiście można znaleźć w dokumentacji. Scope określa nam w jaki sposób mamy podchodzić do umieszczania pobranej dependencji na `classpath`. Przykłady poniżej odnoszą się też do sytuacji, gdy chcemy zbudować wynikowy plik `.jar`. Możemy wtedy zdecydować czy umieszczać zewnętrzne zależności w tym pliku `.jar`, lub czy zależności są nam potrzebne na etapie kompilacji, czy może dopiero w trakcie działania programu. Poniżej wyjaśniam w tabelce, nie podaję też wszystkich dostępnych `scope`.

Scope	Opis
compile	Defaultowy scope, czyli jak nie wybierzemy żadnego to będzie wybrany ten. Dependencje oznaczone jako compile będą dostępne na wszystkich classpathach i na etapie kompilacji i na etapie wykonania programu
provided	Podobny do scope compile, ale określa, że oczekujemy, że zależność jest konieczna na etapie kompilacji. Natomiast w trakcie działania programu oczekujemy, że dependencja zostanie dostarczona przez serwer, na którym uruchamiamy nasz program. Ustawimy też ten zakres jeżeli potrzebujemy dependencji na etapie budowania, ale w trakcie działania już nie. Dependencja w scope provided jest też dostępna w classpath dla testów
runtime	Dependencja nie jest potrzebna na etapie kompilacji, ale jest wymagana w trakcie działania programu.
test	Dependencja nie jest wymagana podczas normalnego działania programu, potrzebujemy jej natomiast jeżeli będziemy pisać testy do naszego programu. Najczęściej używane do bibliotek, które są stosowane tylko do testów.

Jeżeli chodzi o scope `test`, wspomnimy niedługo o testach, wtedy rozjaśni się to bardziej.

Snapshoty

Możliwe jest znalezienie dependencji, które w numerze wersji będą miały dopisek **SNAPSHOT**. Oznaczenie to służy do tego, żeby dać znać korzystającemu z biblioteki, że korzysta z wersji, która nie jest ostateczna i może ulegać zmianie. Czyli jeżeli pobraliśmy zawartość takiej biblioteki dzisiaj i zrobimy to samo jutro, to teoretycznie taka zawartość może się zmienić. Pewnie klasy i metody mogą zostać w tym czasie dodane, a inne mogą zostać usunięte. My również możemy w naszym opisie pliku `pom.xml` stosować dopisek **SNAPSHOT**.

Jeżeli chcemy napisać, że zależy nam na dependencji **SNAPSHOT** definiujemy to w taki sposób:

```
<version>1.0.0-SNAPSHOT</version>
```

Notatki - Maven - Repositories

Spis treści

Repozytoria (Repositories)

Repozytoria `Maven` są miejscami, w których możemy znaleźć pliki `.jar` wraz z metadanymi na temat konkretnych dependencji. W metadanych są m.in. umieszczone informacje na temat zależności od których zależy nasza zależność (masło maślane). W ten sposób `Maven` wie, że jeżeli pobiera dependencję `A` to do jej pełnego działania należy jeszcze pobrać dependencję `B` i `C`. Zależności te będą pobierane do momentu, aż maven nie pobierze wszystkich wymaganych zależności przez zależności, które są przez nas pobierane.

Możemy wyróżnić 3 rodzaje repozytoriów:

- **Local Repository**
- **Central Repository**
- **Remote Repository**

Podane wyżej repozytoria zostały podane w takiej kolejności, bo dokładnie w takiej kolejności `Maven` sprawdza dostępność zależności w repozytoriach. Aby lepiej to zrozumieć, wyjaśnijmy sobie każde z nich.

Local Repository

To repozytorium jest zlokalizowane na komputerze developera piszącego kod. Wszystkie zależności pobierane przez `Maven` zostają zapisane w tym repozytorium. Nie wspomniałem wcześniej (w notatkach), że repozytorium lokalne jest współdzielone między projektami. Czyli jeżeli pracujemy nad 4 projektami i wszystkie z nich zależą od biblioteki `jsoup` to pobierzemy ją tylko raz, a nie 4. Domyślna lokalizacja tego repozytorium to katalog **.m2/repository** w plikach użytkownika, przykładowo na Windows może wyglądać to tak *C:/Users/karol/.m2/repository*. Możemy tę lokalizację zmienić.

Central Repository

Repozytorium centralne jest centralnym miejscem, w którym `Maven` szuka zależności Link. Zależności są pobierane z tego repozytorium jeżeli `Maven` nie znajdzie ich u nas lokalnie na maszynie, czyli w lokalnym repozytorium. Są one następnie pobierane do nas na komputer.

Remote Repository

Repozytorium zdalne jest miejscem, w którym można szukać zależności, ale nie jest to repozytorium centralne. W zasadzie możemy nawet sami wystawić takie repozytorium w Internecie. W praktyce natomiast tego typu repozytoria są to repozytoria wewnątrzfirmowe. Firma, dla której pracujesz posiada w sieci wewnętrznej (niedostępnej normalnie w internecie) umiejscowione repozytorium, z którego powinniśmy pobierać zależności. Możemy też wtedy w takich repozytoriach trzymać dependencje, które nie powinny być dostępne dla świata zewnętrznego, a tylko dla programów tworzonych w danej firmie.

Aby dodać takie repozytorium w pliku `pom.xml` należy dodać tag `repositories` obok taga `dependencies`.

```
<project>
...
    <repositories>
        <repository>
            <id>my-repo1</id>
            <name>my custom repo 1</name>
            <url>http://my1.custom.repo</url>
        </repository>
        <repository>
            <id>my-repo2</id>
            <name>my custom repo 2</name>
            <url>http://my2.custom.repo</url>
        </repository>
    </repositories>
...
</project>
```

Notatki - Maven - Plugins

Spis treści

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'.
```

Notatki - Maven - Fat Jar

Spis treści

Building Fat Jar

`Fat Jar` to pojedynczy plik `.jar`, który zawiera w sobie wszystkie skompilowane klasy z naszego projektu oraz wszystkie skompilowane (zgodnie z konfiguracją) klasy z plików `.jar`, które zostały określone jako dependencje naszego projektu. Plik taki jest o tyle wygodny, że nie musimy wskazywać położenia bibliotek, gdyż są one dołączone w ramach jednego pliku. Nie musimy wtedy wszystkich tych plików wskazywać na `classpath`.

Taki plik `.jar` możemy stworzyć przy pomocy `Maven`. Oczywiście potrzebna jest do tego odpowiednia konfiguracja. W tym celu wykorzystamy plugin `maven-assembly-plugin`. Przykładowa konfiguracja:

```
<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">

    <!-- Tutaj umieszczamy konfigurację, którą widzieliśmy już wcześniej -->
    ...

    <build>
        <finalName>zajavka-title-reader</finalName>
        <plugins>
            <!-- Inne pluginy -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <version>3.1.1</version>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>pl.zajavka.MavenCompilingJsoupExamplesRunner</mainClass>
                        </manifest>
                    </archive>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                </configuration>
                <executions>
                    <execution>
                        <id>make-assembly</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
```

```
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Dzięki określeniu:

```
<descriptorRef>jar-with-dependencies</descriptorRef>
```

mówimy pluginowi, że chcemy stworzyć `Fat Jar`, czyli `Jar with dependencies`. Tag `execution` jest analogiczny do poprzednich przykładów, tym razem jednak dodajemy do tego taga pole `id`. Możemy teraz uruchomić komendę:

```
mvn clean verify
```

Maven wyprodukuje `Fat Jar` w katalogu `target`, który jest domyślnym katalogiem, w którym `Maven` produkuje swoje produkty ☺. Schemat nazewnictwa takiego pliku wygląda następująco:

```
zajavka-title-reader-jar-with-dependencies.jar
```

Jeżeli nie dodalibyśmy taga `execution` z powyższą konfiguracją, należałoby uruchomić utworzenie `Fat Jar` za pomocą komendy:

```
mvn clean package assembly:single
```

Jeżeli natomiast chcemy dać możliwość uruchomienia naszego programu z poziomu pliku `.jar` (inaczej tzw. **executable jar**), należy do konfiguracji dodać fragment:

```
<archive>
    <manifest>
        <mainClass>pl.zajavka.MavenCompilingJsoupExamplesRunner</mainClass>
    </manifest>
</archive>
```

Inaczej przy próbie uruchomienia:

```
java -jar zajavka-title-reader-jar-with-dependencies.jar
```

Dostaniemy poniższy błąd:

```
no main manifest attribute, in zajavka-title-reader-jar-with-dependencies.jar
```

Profile

Profile budowania są wykorzystywane jeżeli chcemy wskazać różnice między sposobami budowania naszej aplikacji w zależności od tego, na które środowisko ma zostać ona wdrożona (DEV, TEST, PROD). Możemy wtedy włączać lub wyłączać konkretne operacje lub ustawienia, w zależności od środowiska, które nas interesuje. Mielibyśmy wtedy np. profil DEV, TEST oraz PROD i każdy profil przygotowywałby zbudowany projekt na dedykowane środowisko. Link do dokumentacji umieszczam tutaj.

Jeżeli chcemy zastosować profil, należy dodać tag profiles jak poniżej:

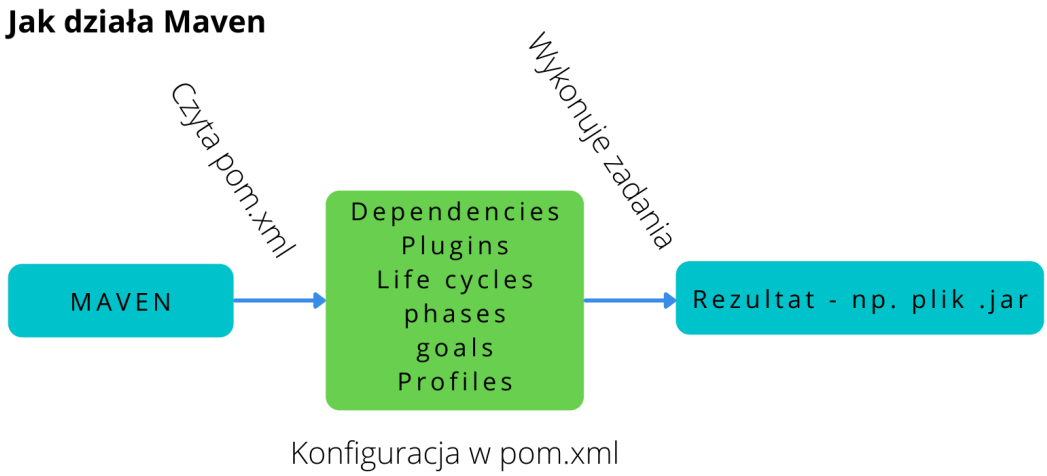
```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    ...
    <profiles>
        <profile>
            <id>TEST</id>
            <build>...</build>
            <modules>...</modules>
            <repositories>...</repositories>
            <dependencies>...</dependencies>
        </profile>
    </profiles>
    ...
</project>
```

Jeżeli następnie chcemy uruchomić komendę maven z wybranym profilem, możemy to zrobić w ten sposób:

```
mvn compile -P TEST
```

Podsumowanie

Podsumowanie działania Maven umieściłem na grafice poniżej.



Obraz 1. Jak działa Maven

Notatki - Gradle - Intro

Spis treści

Zakładam, że na etapie czytania tej notatki masz już podstawową wiedzę odnośnie narzędzi do budowania projektów, która została wyniesiona z materiałów o Maven. Dlatego też w przypadku niektórych terminów nie będziemy schodzić głęboko, bo zostały już wyjaśnione wcześniej.

Czym jest Gradle?

Wiemy już do czego jest zdolny Maven. Powiedzmy sobie również co potrafi i czym jest **Gradle**.

Gradle jest narzędziem stosowanym do zarządzania budowaniem aplikacji łącznie z automatyczną obsługą zależności i bibliotek. Obsługuje repozytoria Maven aby te zależności pobierać. Kod piszemy tutaj przy wykorzystaniu **DSL** (Domain Specific Language), którym jest Groovy. Nie musimy całkowicie znać języka Groovy, aby móc tego narzędzia używać. Wiele osób, które stosuje Gradle nie są programistami języka Groovy.

Swoją drogą, wielu osobom też mieszają się te dwie nazwy: Gradle i Groovy. Może żeby było łatwiej to zapamiętać, Gradle - do budowania jak szpadle, Groovy - konfigurację mówi. Albo wymyśl sobie własne rymy 😁.

Zanim jednak przejdę do porównania Maven i Gradle ze sobą, opowiem trochę o tym jak działa Gradle.

Obraz 1. Gradle logo. Źródło: https://commons.wikimedia.org/

Plik z konfiguracją (build script)

Konfiguracja analogiczna do `pom.xml` jest opisywana w plikach `build.gradle` przy wykorzystaniu języka `Groovy`. Jeżeli chcemy uruchomić jakieś polecenie przy wykorzystaniu narzędzia `Gradle`, należy użyć komendy `gradle`.

Projects oraz Tasks

Podczas gdy w `Maven` mówiliśmy o `lifecycle`, `phase` i `goal`, tutaj będziemy mówili o `project` oraz `task`.

Patrząc całościowo, projekt może być reprezentacją końcowego pliku `.jar`, który chcemy wytworzyć, lub końcowej aplikacji webowej, nad którą pracujemy. Projekt może również służyć do reprezentacji końcowego pliku `.zip`. Abstrakcyjne rzecz mówiąc, projekt może być czymś co należy zbudować, lub czymś co należy zrobić. Inaczej mówiąc projekt jest czymś co na koniec chcemy wytworzyć. Aplikacje w praktyce mogą składać się z kilku modułów (czego nie poruszamy), taki moduł jest wtedy projektem w rozumieniu `Gradle`. Do tego projekt składa się z zadań (`task`).

`Task` natomiast, jest to część pracy jaką należy wykonać aby przyczynić się do powstania końcowego `buildu`. Zadaniem może być wykonanie kompilacji kodu, wygenerowanie dokumentacji `Javadoc`, stworzenie pliku `.jar` lub umieszczenie zależności w repozytorium.

Definiowanie własnego taska

Poniżej umieszczam bardzo prosty plik `build.gradle` z własnym taskiem:

```
task ourFirstTask {
    doLast {
        println 'zajavka task'
    }
}
```

Jeżeli teraz chcemy uruchomić powyższe zadanie, należy (w folderze, gdzie mamy zlokalizowany plik `build.gradle`) wykonać komendę:

```
gradle ourFirstTask
```

Na ekranie zostanie wtedy wydrukowane coś podobnego do:

```
Starting a Gradle Daemon, 1 incompatible and 1 stopped Daemons could not be reused, use --status for
details

> Task :ourFirstTask
zajavka task

BUILD SUCCESSFUL in 3s
1 actionable task: 1 executed
```

Jeżeli chcemy ograniczyć się do wydrukowania na ekranie tylko wiadomości, która nas interesuje,

możemy wykonać to samo polecenie z flagą `-q` (quiet). Powoduje to wtedy, że `Gradle` jest mniej "gadatliwy":

```
gradle ourFirstTask -q
```

Jeżeli chcemy przykładowo sprawdzić położenie plików `Gradle` na naszej maszynie, możemy dopisać do tego samego pliku poniższy task i go wywołać:

```
task getHomeDir {
    doLast {
        println gradle.gradleHomeDir
    }
}
```

Taski domyślne

Oprócz tego, że możemy napisać swoje własne taski, możemy również zobaczyć jakie domyślne zadania `Gradle` ma do zaoferowania. W tym celu należy wykonać komendę `gradle tasks -q`. Zobaczysz wtedy na ekranie napisy takie jak **Build Setup tasks** lub **Help tasks**. Są to oznaczenia grupy tasków.

Grupowanie tasków

Jeżeli dołożymy do tego komendę `gradle tasks -q --all` zobaczymy również nasze taski, które nie są przypisane do żadnej grupy. Możemy je przypisać do grupy w ten sposób:

```
task getHomeDir {
    group 'zajavka'
    doLast {
        println gradle.gradleHomeDir
    }
}
```

Jeżeli teraz uruchomimy komendę `gradle tasks -q -all` zobaczymy na ekranie również grupę **zajavka tasks**.

Struktura tasków

Sam task składa się z różnych faz jego wykonania. Możemy wyróżnić fazę konfiguracji, która jest definiowana poza np. fragmentem `doLast`. Dodanie grupy jest przykładem fazy konfiguracji. Oprócz fazy konfiguracji możemy wyróżnić `doFirst` oraz `doLast`. Wiedząc o tym, możemy napisać taką konfigurację:

```
task ourSecondTask {
    group 'zajavka'
    description 'zajavka description'
    println 'Always printed'
}

task ourThirdTask {
    group 'zajavka'
```

```
    doFirst {
        println 'doFirst'
    }

    doLast {
        println 'doLast'
    }
}
```

Jeżeli teraz uruchomimy polecenie:

```
gradle ourThirdTask -q
```

To na ekranie wydrukuje się napis:

```
Always printed
doFirst
doLast
```

Własne taski domyślne

Dołóżmy do tego, że `Gradle` pozwala nam również zdefiniować własne taski domyślne w pliku `build.gradle`, które zostaną uruchomione, jeżeli nie określimy zadania, które ma zostać uruchomione. Przykład poniżej:

```
defaultTasks 'ourFirstTask', 'getHomeDir'

task ourFirstTask {
    group 'zajavka'
    doLast {
        println 'zajavka task'
    }
}

task getHomeDir {
    group 'zajavka'
    doLast {
        println gradle.gradleHomeDir
    }
}
```

Jeżeli teraz uruchomimy komendę:

```
gradle -q
```

Na ekranie zostanie wydrukowane:

```
zajavka task
```

```
C:\Users\karol\.gradle\...\gradle
```

Zależności między taskami

Możemy również określać zależności między zadaniami. Spójrz na konfigurację poniżej:

```
task clean {
    doLast {
        println 'Pretending to clean temporary files'
    }
}

task compile {
    doLast {
        println 'Pretending to compile Java code'
    }
}

task afterCompile(dependsOn: 'compile') {
    doLast {
        println 'Running after compile'
    }
}

task afterClean {
    doLast {
        println 'Running after clean'
    }
}

afterClean.dependsOn clean
```

Spróbuj teraz uruchomić kolejno poniższe polecenia i zobacz jaki będzie rezultat na ekranie:

```
gradle clean -q
gradle compile -q
gradle afterClean -q
gradle afterCompile -q
```

Taski `clean` i `compile` uruchamiają tylko same siebie. `AfterClean` oraz `afterCompile` uruchamiają najpierw odpowiednio `clean` i `compile`, bo tak zostało to zdefiniowane w pliku `build.gradle`.

Notatki - Gradle - Java

Spis treści

Zakładam, że na etapie czytania tej notatki masz już podstawową wiedzę odnośnie narzędzi do budowania projektów, która została wyniesiona z materiałów o Maven. Dlatego też w przypadku niektórych terminów nie będziemy schodzić głęboko, bo zostały już wyjaśnione wcześniej.

Gradle wrapper

Jeżeli chcielibyśmy móc uruchamiać zadania `Gradle` nie mając zainstalowanego `Gradle` lokalnie, możemy wykorzystać **Gradle** `wrapper`. W momencie, gdy uruchamiamy `Gradle` stosując **wrapper**, konkretna wersja `Gradle` jest automatycznie pobierana i używana do wykonania danego builda. Pozwala to na niezależność od zainstalowanej wersji `Gradle` na maszynie, na której pracujemy. Załóżmy, że mamy zainstalowanego `Gradle` w wersji 6.8 i potrzebujemy uruchomić build dla 3 projektów, które stosują inne wersje `Gradle`. Dzięki `Gradle` **wrapper** możemy wykonać buildy z różnymi wersjami `Gradle` w różnych projektach. Aby faktycznie wygenerować pliki wrappera, musimy mieć zainstalowany `Gradle` na naszym systemie operacyjnym.

Aby wygenerować pliki wrappera można zastosować poniższą komendę. Możemy też nie podawać wersji, `Gradle` weźmie wtedy najnowszą.

```
gradle wrapper --gradle-version <wersja>
```

Zamiast `<wersja>` możemy wpisać np. `7.4` lub `7.6`.

Wygenerowane wtedy zostaną pliki:

```
<project folder>
    gradlew
    gradlew.bat
    gradle
        wrapper
            gradle-wrapper.jar
            gradle-wrapper.properties
```

Uruchamiamy wtedy taski `Gradle` przy wykorzystaniu polecenia:

```
./gradlew build
// lub jak nam nie zadziała, to zwyczajnie
gradlew build
```

Gradle Wrapper jest świetną możliwością, która pozwala nam dostosowywać wersję Gradle do konkretnego projektu. Jeżeli masz zainstalowany Gradle globalnie (na cały system operacyjny) w wersji np. 7.3, to możesz w trzech różnych projektach korzystać z różnych wersji Gradle np. 7.4, 7.5 oraz 7.6. Wszystko jest kwestią wartości ustawionej w pliku `gradle-wrapper.properties`. Plik ten jest wtedy dedykowany dla konkretnego projektu.

Plik gradle-wrapper.properties

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-7.6-bin.zip
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
```

Budowanie projektów Java

`Gradle` zawiera wbudowany domyślnie plugin umożliwiający kompilację kodu źródłowego Javy, uruchomienie testów (o tym później), tworzenie plików `.jar` oraz generowanie dokumentacji `Javadoc`. Ponownie odwołam się to terminu **convention over configuration**. W tym przypadku stosowanie konwencji w projekcie budowanym przy pomocy `Gradle` oznacza zachowanie określonej struktury projektu. Plugin Java zakłada istnienie folderów:

- **src/main/java** - w którym umieszczamy pliki źródłowe aplikacji Java
- **src/test/java** - w którym umieszczamy pliki źródłowe **testów** naszej aplikacji

Jeżeli będziemy trzymać się tej konwencji, jedyne co jest nam potrzebne aby określić konfigurację do budowania projektu Java to stworzenie pliku `build.gradle` z poniższą linijką:

```
apply plugin: 'java'
```

Więcej o możliwościach tego pluginu można poczytać tu.

Po stworzeniu konfiguracji zawierającej tę linijkę, uruchom komendę `gradle tasks`. Możesz zwrócić uwagę, że zostało dodanych dużo nowych tasków, które są specyficzne dla Javy. Dzięki temu będziemy teraz mogli wykorzystać task służący do budowania. Aby w tym momencie uruchomić build, należy wywołać komendę `gradle build`.

Task `gradle build` kompiluje kod, wykonuje testy (o których powiemy później) oraz tworzy wynikowy plik `.jar`. Aby na tym etapie zobaczyć rezultaty wykonanej komendy należy spojrzeć do folderu `build`. W środku znajdziemy następujące katalogi:

- **classes** - w środku znajdziemy skompilowane pliki `.class`. Tutaj należy szukać plików `.class`

odpowiadających naszym plikom źródłowym z katalogu `src/main/java`

- **reports** - w środku znajdziemy wygenerowane raporty, które są tworzone np. przy uruchamianiu testów naszego programu. Na ten moment jednak nie dodaliśmy żadnych testów do projektu
- **libs** - w środku znajdziemy pliki `.jar` utworzone w trakcie trwania buildu. Zwróć uwagę, że utworzony plik `.jar` nie zawiera wersji

Wersja pliku jar

Jeżeli chcemy określić wersję pliku `.jar`, który wygenerowaliśmy, należy dodać do pliku `build.gradle` następującą linijkę:

```
apply plugin: 'java'
version = '0.1.0'
```

Wersja Javy

Tak jak w przypadku `Maven`, mogliśmy określić parametry takie ja `source` i `target` służące do podania kompatybilności z wersją Javy odpowiednio na etapie kompilacji lub uruchamiania programu, tak samo możemy zrobić to tutaj. Do pliku `build.gradle` możemy dodać parametry takie jak:

- **sourceCompatibility** - cytując dokumentację - Java version compatibility to use when compiling Java source. Default value: version of the current JVM in use.
- **targetCompatibility** - cytując dokumentację - Java version to generate classes for. Default value: sourceCompatibility.

```
apply plugin: 'java'
version = '0.1.0'
sourceCompatibility = '17'
```

Skoro dokumentacja mówi, że wartość domyślna parametru `targetCompatibility` to `sourceCompatibility` to ustawiamy tylko wartość parametru `sourceCompatibility`.

Jeżeli natomiast chcemy zobaczyć która wersja `Gradle` wspiera którą wersję Javy, można to zrobić w dokumentacji.

Executable jar

Co zrobić jeżeli chcemy aby nasz plik `.jar` był uruchamialny? Czyli chcemy mieć możliwość uruchomienia naszego programu bezpośrednio z pliku `.jar`. W paczce `gradle` w katalogu `src/main/java` stwórzmy poniższy plik:

```
package pl.zajavka;

public class GradleExamples {

    public static void main(String[] args) {
```

```
        System.out.println("Gradle examples");
    }
}
```

Do tego w pliku `build.gradle` określmy poniższą konfigurację:

```
apply plugin: 'java'

jar {
    archivesBaseName = 'myBaseName'
    manifest {
        attributes 'Main-Class': 'pl.zajavka.GradleExamples'
    }
}
```

Jeżeli teraz uruchomimy komendę `gradle clean build`, to możemy spróbować uruchomić nasz program `.jar` za pomocą komendy `java -jar <nazwa_pliku_jar>.jar`.

Taski w 'gradle build'

Na stronie z dokumentacją pluginu o tutaj znajduje się obrazek o ten przedstawiający jakie zadania są wykonywane przed zadaniem `build`, w ramach zależności między taskami. Pokazywaliśmy to wcześniej. Jeżeli spojrzymy pod ten adres zobaczymy opis tych zadań i ich wzajemne zależności. Na tej podstawie widać, że zanim wykonany zostanie task `build`, wcześniej muszą się wykonać też inne, takie jak chociażby `check` i `assembly`.

Notatki - Gradle - Dependencies

Spis treści

Zakładam, że na etapie czytania tej notatki masz już podstawową wiedzę odnośnie narzędzi do budowania projektów, która została wyniesiona z materiałów o Maven. Dlatego też w przypadku niektórych terminów nie będziemy schodzić głęboko, bo zostały już wyjaśnione wcześniej.

Dependencies

`Gradle` podobnie jak `Maven` wspomaga nas w zarządzaniu zależnościami na naszym `classpath`. Oczywiście tak jak `Maven` - `Gradle` pobiera te zależności za nas. A jak zintegrujemy `Gradle` z Intellij to zależności te mogą być pobierane przy wykorzystaniu przycisku do odświeżenia (tak jak pokazywałem to w przypadku `Maven`).

Znamy już znaczenie terminów takich jak `groupId`, `artifactId` oraz `version`. Zależność w `Gradle` jest definiowana również przy wykorzystaniu tych 3 słów kluczowych. Wygląda to wtedy w ten sposób:

```
group: '<groupId>', name: '<artifactId>', version: '<version>'
```

Lub w wersji skróconej:

```
groupId:artifactId:version
```

Aby dodać zależność, albo zależności do naszego projektu, należy dodać sekcję `dependencies`, przykładowo:

```
plugins {
    id 'java'
}

group = 'pl.zajavka'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'
```

```
repositories {
    mavenCentral()
}

dependencies {
    implementation group: 'org.jsoup', name: 'jsoup', version: '1.14.3'
    // lub w wersji skróconej
    // implementation 'org.jsoup:jsoup:1.14.3'
    testCompile 'junit:junit:4.12'
}

jar {
    manifest {
        attributes 'Main-Class': 'gradle.GradleExamples'
    }
}
```

Pojawiły nam się 3 nowe bloki:

- **plugins** - umożliwia nam używanie kilku pluginów jednocześnie. Zastąpiło fragment `apply plugin: 'java'`.
- **repositories** - określa, że dependencje mają być zaciągane z centralnego repozytorium Maven.
- **dependencies** - blok, w którym możemy definiować zależności, które będą nam potrzebne na różnych `classpathach` (compile vs runtime classpath, pamiętasz?).

Biblioteką `junit` na razie się nie przejmuj. Została ona tutaj dodana żeby pokazać w jaki sposób zapisujemy wiele bibliotek.

Blok dependencies

W bloku dependencies pojawiły się słówka `implementation` oraz `testCompile`. Są one używane do określenia typów konfiguracji konkretnych dependencji. Upraszczając, określają one w którym momencie dana biblioteka ma być używana na **classpath**. Podstawowe typy konfiguracji, które będziemy stosować (nie wymieniam tu wszystkich możliwych):

- **compileOnly** - zależność jest używana tylko na 'compile classpath'.
- **runtimeOnly** - zależność jest używana tylko na 'runtime classpath'.
- **implementation** - zależność jest używana na obu classpath.
- **testCompileOnly** - zależność jest używana tylko na 'compile classpath', ale tylko w testach.
- **testRuntimeOnly** - zależność jest używana tylko na 'runtime classpath', ale tylko w testach.
- **testImplementation** - zależność jest używana na obu classpath, ale tylko w testach.

Mamy również konfiguracje, które będziesz spotykać w źródłach w internecie, natomiast na moment pisania tego tekstu są one **deprecated** (przestarzałe, kandydat usunięcia w przyszłych wersjach). W nowszych wersjach (np. 7.3) nawet nie możemy ich już zastosować bo są usunięte. Są to:

- **compile** - zamiast tego typu powinniśmy używać `implementation`.
- **runtime** - zamiast tego typu powinniśmy używać `runtimeOnly`.
- **testCompile** - zamiast tego typu powinniśmy używać `testImplementation`.

- **testRuntime** - zamiast tego typu powinniśmy używać `testRuntimeOnly`.

Jeżeli ktoś jest zainteresowany poznaniem większej ilości, umieszczam link.

W dokumentacji `Gradle` można jednocześnie znaleźć informacje dotyczące migracji plików `Gradle` pomiędzy kolejnymi wersjami. Dla zainteresowanych umieszczam link.

Gdzie Gradle przechowuje pobrane biblioteki?

W przypadku `Maven` mówiliśmy, że pobrane biblioteki są przetrzymywane w katalogu `.m2/repository`. W przypadku `Gradle`, pobrane zależności są przetrzymywane w katalogu `.gradle/caches/modules-2/files-2.1`. Nie są one współdzielone między `Maven` a `Gradle`, są w innych katalogach.

Blok repositories

Wcześniejszy przykład pokazał jak użyć repozytorium **Maven Central**. Mechanizm repozytoriów został omówiony w materiałach o `Maven`. Oprócz Maven Central możemy również określić swoje własne repozytoria. Przykładowa konfiguracja:

```
repositories {
    maven {
        url "http://repo.zajavka.pl/maven2"
    }
}
```

Możemy również dodać kilka repozytoriów jednocześnie:

```
repositories {
    maven {
        url "http://repo.zajavka.pl/maven2"
    }
    maven {
        url "http://repo.otherwebpage.com/maven2"
    }
}
```

Natomiast możemy w konfiguracji dodać również taki zapis:

```
repositories {
    mavenCentral()
}
```

Nie zostało to pokazane w materiałach, spróbuj usunąć pobraną bibliotekę z katalogu `.gradle/caches/modules-2/files-2.1`, następnie usuń z `gradle.build` wpis dotyczący `mavenCentral()` i zobacz co się stanie z zależnościami `jsoup` w pliku `Main.java`.

Fat Jar

Z racji, że wcześniej wspominaliśmy o koncepcji `Fat Jar`, spróbujmy napisać konfigurację `Gradle`, która przygotuje **jar-with-dependencies**.

```
plugins {
    id 'java'
}

group = 'pl.zajavka'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

repositories {
    mavenCentral()
}

dependencies {
    implementation group: 'org.jsoup', name: 'jsoup', version: '1.14.3'
}

jar {
    archivesBaseName = 'myBaseName'
    manifest {
        attributes 'Main-Class': 'gradle.GradleExamples'
    }
    from {
        configurations.runtimeClasspath.collect { it.isDirectory() ? it : zipTree(it) }
    }
}
```

Po dodaniu powyższej konfiguracji i wykonaniu komendy `gradle build`, spróbuj rozpakować plik `.jar` i zobacz jego zawartość.

Gradle Settings

Oprócz pliku `build.gradle` możemy również stworzyć plik `settings.gradle`. W praktyce jest to używane wtedy gdy nasza aplikacja składa się z kilku modułów. My cały czas obracamy się w zakresie aplikacji, które mają tylko jeden moduł. W takim przypadku, możemy w pliku `settings.gradle` dodać taki wpis:

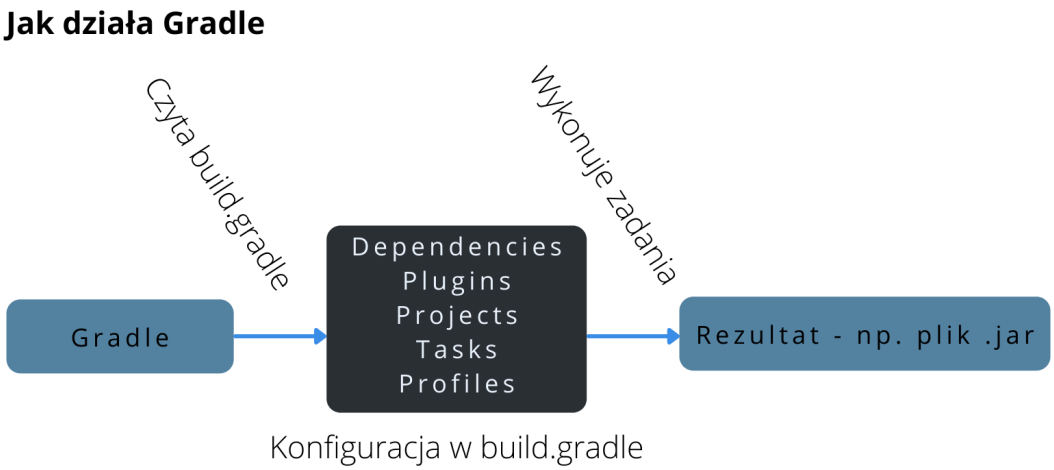
```
rootProject.name = 'java-gradle-examples'
```

Jeżeli natomiast chcielibyśmy zobaczyć z ilu modułów składa się nasza aplikacja, możemy wykonać polecenie:

```
gradle projects
```

Podsumowanie

Podsumowanie działania `Gradle` umieściłem na grafice poniżej.



Obraz 1. Jak działa Gradle

Notatki - Gradle vs Maven

Spis treści

Zakładam, że na etapie czytania tej notatki masz już podstawową wiedzę odnośnie narzędzi do budowania projektów, która została wyniesiona z materiałów o Maven. Dlatego też w przypadku niektórych terminów nie będziemy schodzić głęboko, bo zostały już wyjaśnione wcześniej.

Gradle czy Maven? Maven czy Gradle?

Jak zaczniesz szukać, które z tych narzędzi jest lepsze (cokolwiek to znaczy) natkniesz się również na narzędzie **Ant**. Celowo nie poruszałem tej tematyki, bo widziałem je w praktyce tylko raz, dawno temu.

Maven

`Maven` powstał po `Ant` (czasowo, historycznie) i korzystał z pewnych konwencji oraz rozwiązań stosowanych w `Ant`. W `Maven` konfiguracja jest określana w pliku **XML** i korzystamy tutaj z podejścia **convention over configuration**. Dzięki temu mamy dostępne dużo predefiniowanych operacji, które są dostępne po dodaniu odpowiednich pluginów. Nie musimy tych operacji pisać samodzielnie. `Maven` zapewnia nam predefiniowane `phases` oraz `goals`. W podejściu stosowanym przy konfiguracji `Maven` możemy skupić się na określeniu co ma się stać w naszym buildzie, nie musimy opisywać w jaki sposób ma się to stać. Często można spotkać się ze stwierdzeniem, że `Maven` jest frameworkiem do uruchamiania pluginów, z racji, że cała jego praca jest wykonywana przy wykorzystaniu pluginów.

Framework można na tym etapie rozumieć jak dużą bibliotekę, która w uproszczeniu może służyć za szkielet aplikacji.

`Maven` stał się bardzo popularnym narzędziem, ze względu na swoją standaryzację. Sprawy komplikują się gdy pojawia się potrzeba dużej customizacji, czyli wprowadzania mocno niestandardowych rozwiązań. Jeżeli natomiast korzystamy z `Maven` w sposób standardowy, jego konfiguracja jest prosta i przejrzysta. W praktyce natomiast potrafi mocno spuchnąć, często plik `pom.xml` potrafi być bardzo duży, mieć bardzo dużo linijek. Wynika to chociażby z ilości dodanych pluginów czy zależności. Dla porównania, mówi się, że `Ant` pozwalał na dużą elastyczność w określeniu konfiguracji budowania aplikacji, ale przez to traciliśmy standaryzację.

Gradle

Adresując potrzebę elastyczności i standaryzacji jednocześnie powstało narzędzie zwane `Gradle`. `Gradle` powstał przy stosowaniu koncepcji używanych w `Ant` i `Maven`. `Gradle` nie stosuje plików **XML**, konfiguracja w `Gradle` jest pisana w `DSL`, którym może być np. `Groovy` (może być też `Kotlin`). Dzięki temu, często pliki konfiguracyjne `Gradle` potrafią być mniejsze (w kontekście ilości tekstu) niż w przypadku `Maven`. W przypadku `Gradle` poruszamy się w nomenklaturze zadań (task), a nie faz (phase) i celów (goal). `Gradle` jest przykładowo stosowany jeżeli będziemy pisać w Javie środowisku `Android`.

W `Gradle` zostały wprowadzone koncepcje **incremental compilation** oraz **compile avoidance**. Obie te koncepcje w uproszczeniu zakładają, że `Gradle` jest w stanie sprawdzić, które części naszego kodu uległy zmianie pomiędzy buildami i w konsekwencji kompilować tylko te pliki, które faktycznie uległy zmianie. Często nie ma potrzeby kompilować całego projektu ponownie, bo przecież nie zmieniamy zawsze wszystkich plików. Przekłada się to na skrócenie czasu buildu.

Jeżeli zaczniemy wyszukiwać w internecie testy porównujące czas trwania buildu `Maven` vs `Gradle`, źródła często podają, że `Gradle` radzi sobie o wiele szybciej niż `Maven`.

W praktyce buildy nie trwają 30 sekund jak w naszych przykładach, tylko potrafią trwać często kilka - kilkanaście minut. Wtedy staje się bardzo istotna kwestia możliwości skrócenia czasu trwania takiego buildu, oszczędzamy wtedy czas, podczas którego 'nic nie robimy' ☺.

Zarówno `Maven` i `Gradle` mogą korzystać z Centralnego Repozytorium Maven. Pomagają zarządzać zależnościami, dają nam możliwość kompilacji, tworzenia plików `.jar`, uruchamiania testów oraz generowania dokumentacji. Pojawia się zatem pytanie, który z nich wybrać. Można się wtedy kierować różnymi czynnikami.

`Gradle` i jego **DSL** jest bardziej zwięzły niż **XML**, co powoduje, że plik z konfiguracją jest 'czystszy'. Często mówi się natomiast, że jeżeli potrzebujemy standardowego narzędzia, bez fajerwerków, możemy spokojnie korzystać z `Maven`. Jeżeli zależy nam na customizacji procesu budowania, tworzeniu własnych zachowań, wtedy polecany jest `Gradle`.

Truizm o aktualizacji wersji

Możesz zacząć się zastanawiać, jak to wygląda w praktyce z aktualizacją wersji bibliotek, albo narzędzi, albo samej Javy. Firmy robią to często, czy sporadycznie? Odpowiedź na to pytanie, to ulubiona odpowiedź Karola: *"to zależy"*.

Działa? To zostaw!

Spotkasz na swojej ścieżce zawodowej firmy, w których: *"jak działa to nie ruszaj, chyba, że jest to absolutnie konieczne"*. To, że na rynku dostępna jest najnowsza Java, albo najnowszy Spring, wcale nie oznacza, że będą one wykorzystywane przez Ciebie w praktyce. Niektóre firmy przetrzymują moment aktualizacji wersji narzędzi do momentu, gdy jest to absolutnie konieczne (np. kończy się wsparcie danej wersji Javy, czym jest wsparcie będzie później). Jakie są konsekwencje takiego podejścia?

W praktyce dużo korzysta się z zewnętrznych bibliotek, co jest zrozumiałe. Lepiej jest skorzystać z

rozwiązania, które wypracował ktoś, kto zna się na danym temacie lepiej niż my sami, niż wymyślać koło od nowa. Konsekwencje tego są natomiast takie, że jeżeli w danej bibliotece została wprowadzona jakaś dziura bezpieczeństwa (czyli potencjalnie nasza aplikacja jest wtedy narażona na ataki), to potencjalny atakujący, jeżeli dowie się, z jakich narzędzi korzystamy, to będzie też wiedział, jak można złamać zabezpieczenia.

To, co teraz opisuję, jest bardzo ogólne. W praktyce takie sytuacje się zdarzały, np. Log4j security vulnerability.

Co wtedy? Wszystkie ręce na pokład i aktualizujemy podatne biblioteki do najnowszych wersji. Gorzej, jeżeli okaże się, że temat aktualizacji bibliotek był tak mocno zaniedbany, że nie można tego zrobić prosto, szybko i wygodnie. W praktyce tak też się zdarza. Wynika to z tego, że biblioteki często mogą być niekompatybilne ze sobą (zobaczysz w praktyce ile tego jest) i zmiana wersji jednej biblioteki może wymuszać zmianę wersji kolejnych 20. Dlatego osobiście wolę drugie podejście.

Jest nowa wersja? Bierem!

Jeżeli tylko wypuszczona została najnowsza wersja biblioteki/narzędzia, aktualizujemy to w projekcie. Oczywiście trzeba to robić z głową, bo najnowsze wersje narzędzi cierpią na choroby wieku dziecięcego. Co więcej, jeżeli zaktualizujesz jakąś bibliotekę, to może się okazać, że cały projekt przestaje działać, bo pozostałe biblioteki nie mają dostępnych nowszych wersji i nie są w stanie współpracować z tą jedną zaktualizowaną. No proza codziennego życia programisty ...

Trzeba jednak pamiętać, że w takim podejściu jesteśmy bardziej "na bieżąco" i jeżeli wystąpi krytyczna potrzeba zwiększenia wersji bibliotek albo narzędzi, to taka aktualizacja przebiegnie bardziej bezboleśnie. Będzie to wynikało z tego, że aktualizujemy często i rozwiązujemy częściej mniejsze problemy. Lepiej jest rozwiązywać częściej mniejsze problemy niż rzadziej ogromne.

To może pisać wszystko samodzielnie?

Korzystanie z zewnętrznych bibliotek wiąże się z potencjalnym ryzykiem security. Z jednej strony ktoś wykonał za Ciebie dużo pracy i nie musisz pisać kodu odpowiedzialnego za coś, bo ktoś już to zrobił i możesz to wykorzystać w postaci biblioteki. Z drugiej strony, jeżeli twórca biblioteki popełnił błąd, albo wprowadził potencjalną podatność security, ta podatność będzie też dostępna w Twoim kodzie. To może pisać wszystko samodzielnie?

Od razu odpowiem, że w praktyce nie wymyśla się koła na nowo. Po to powstały narzędzia. Jeżeli okaże się, że w Javie jest zaszyta jakaś podatność security, to język programowania też będziesz wymyślać na nowo? ☺

Korzystając z narzędzi trzeba jednak pamiętać, że potencjalnie mogą one zawierać błędy. Z drugiej jednak strony, piszą to ludzie, którzy się w tym specjalizują. Biblioteki takie są używane w wielu projektach na świecie (czyli nie tylko my mamy taki problem), zatem rozwiązanie które wykorzystujemy jest dobrze przetestowane, bo korzystają z niego często ludzie na całym świecie. Jeżeli zostanie tam znaleziony błąd, jest on szybko naprawiany i możemy pobrać nowszą wersję.

Jeżeli pisalibyśmy wszystko sami, to:

- po pierwsze musielibyśmy wszystkie błędy rozwiązywać sami (korzystając z bibliotek ktoś robi to za

nas),

- po drugie, samodzielnie moglibyśmy wprowadzić więcej dziur security niż potencjalne dziury w bibliotekach,
- po trzecie, testowanie leżałoby na naszych barkach.

Korzystając z bibliotek dostajemy gotowy, działający, przetestowany, sprawdzony i używany przez całą społeczność programistów kod.

Notatki - Gradle 8

Spis treści

Kompatybilność Gradle i Java

Zaczniemy od tematu, który jest dosyć istotny, gdy zaczniesz zabawę ze zmienianiem wersji Gradle i Javy. Spójrz na tabelkę dostępną tutaj. W tym miejscu znajdziesz informacje o tym, jaka wersja Java wymaga jakiej minimalnej wersji Gradle.

Wiem, że na tym etapie nie omawiamy jeszcze wersji Java i nowości w nich wprowadzanych, ale z racji, że Java jest kompatybilna wstecznie, to możesz pracować u siebie np. z Java 19 (pomimo, że nie poruszyliśmy jeszcze technik, które zostały wprowadzone np. w Java 17) i nadal realizować razem materiały. Oczywiście jakieś drobne zmiany mogą być w kolejnych Javach wprowadzane, przykładowo w Java 19, konstruktor `new Locale` stał się `@Deprecated`. Ale o co w tym chodzi i czym jest `@Deprecated` dowiesz się w dedykowanym warsztacie. Na ten moment istotne jest tylko to, że jeżeli masz zainstalowaną np. Javę 19, to musisz mieć minimalnie Gradle 7.6, co jest zaprezentowane w wymienionej tabelce. Jeżeli zaczniesz coś tutaj kombinować, będziesz mieć błędy, proste ☺.

Gradle 8

Team Gradle wypuścił Gradle 8 w lutym 2023. W jaki sposób można zweryfikować, czy zostały wprowadzone zmiany, które są znaczące (psujące) z naszego punktu widzenia?

Sprawdź istniejące projekty

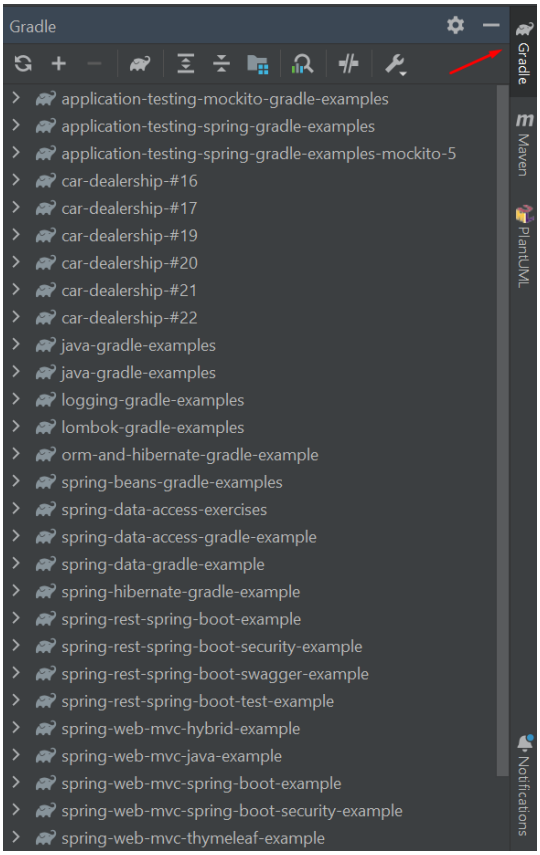
Najlepiej byłoby wziąć jakiś bardziej zaawansowany projekt, który został przygotowany przy wykorzystaniu Gradle, zmienić wersję Gradle w pliku `gradle-wrapper.properties`, spróbować uruchomić `clean` oraz `build` i zobaczyć, czy coś przestało działać.

Plik gradle-wrapper.properties

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
```

```
distributionUrl=https\://services.gradle.org/distributions/gradle-8.0.2-bin.zip
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
```

Ja (Karol) jestem w trochę bardziej komfortowej sytuacji, gdyż mam przygotowanych bardzo dużo mini projektów z konfiguracjami Gradle, które służą mi do przygotowywania materiałów Zajavka. Ty będziesz zapoznawać się z tymi projektami po kolei w kolejnych warsztatach. Na potwierdzenie, tak wygląda to w moim Intellij:

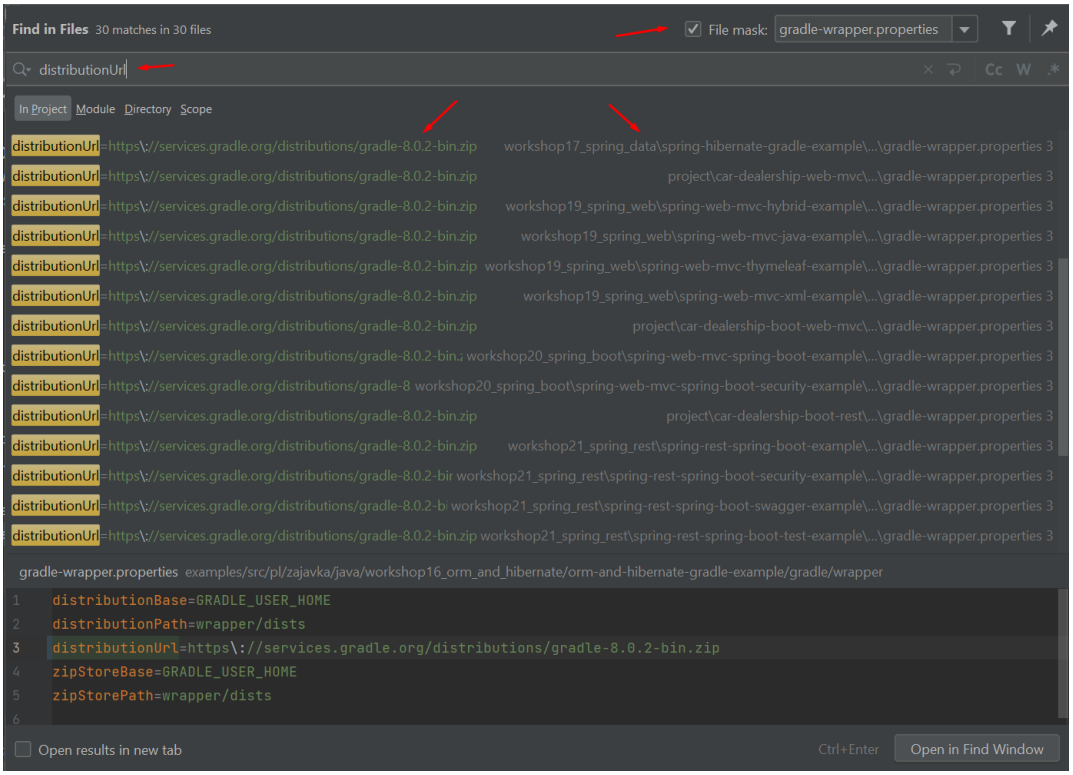


Obraz 1. Intellij Gradle Projects

Zobaczysz tutaj wiele niezrozumiałych jeszcze na tym etapie nazw, ale spokojnie, wszystkie te zagadnienia będą wyjaśniane na przestrzeni kolejnych warsztatów.

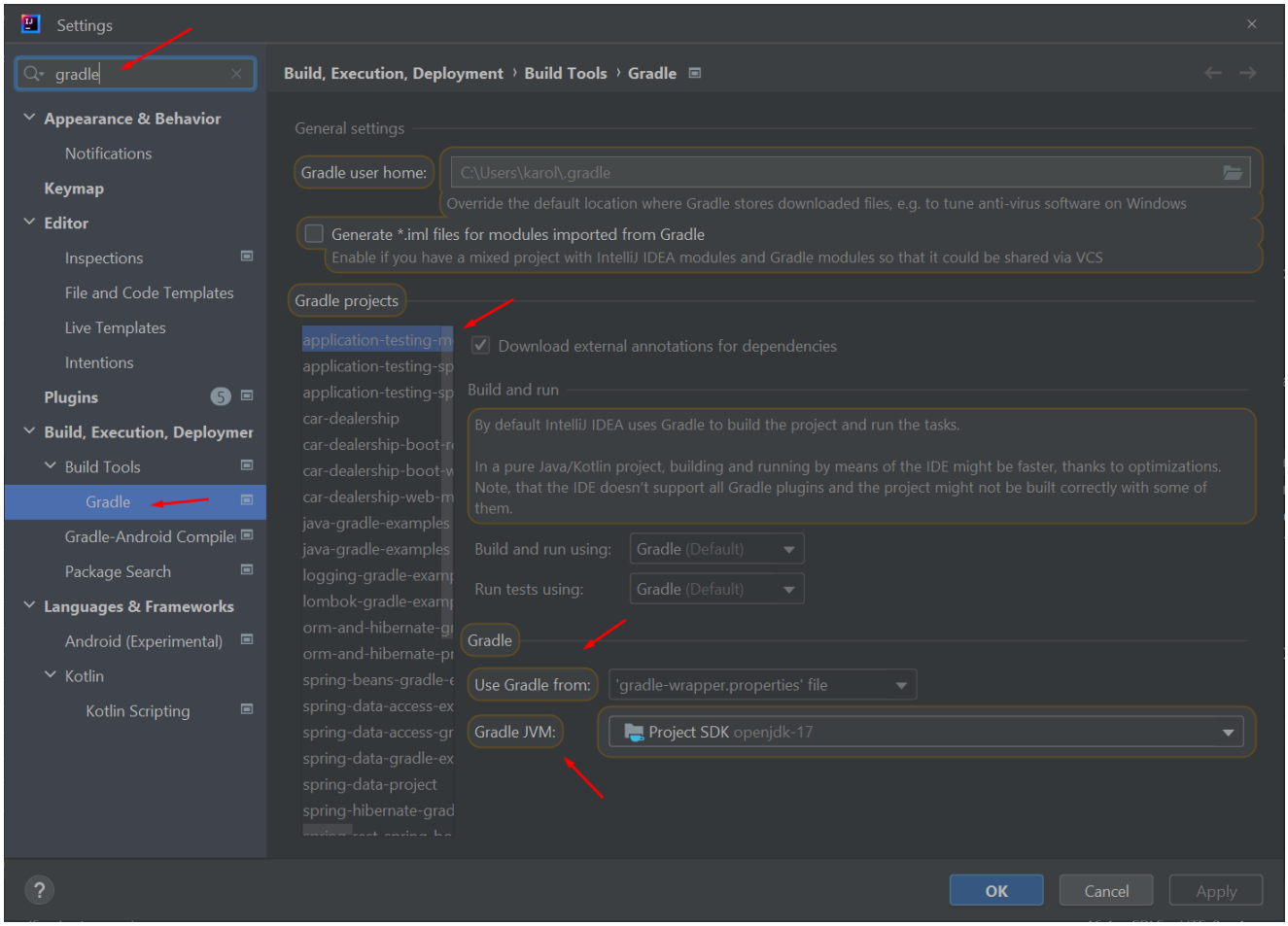
Po co to pokazuję? Żeby pokazać Ci, że testowałem Gradle 8 na wielu z tych konfiguracji Gradle i w żadnej z nich nie wystąpił błąd po zmianie wersji na Gradle 8.

A w jaki sposób to testowałem? Najpierw zamieniłem wersję Gradle we wszystkich plikach `gradle-wrapper.properties`. Każdy z wymienionych projektów ma swój plik `gradle-wrapper.properties`. Wykorzystałem do tego celu skrót *CTRL* + *SHIFT* + *R*. Następnie mogę sprawdzić, wyszukując w całym moim projekcie Intellij, jaka jest ustawiona wartość parametru *distributionUrl*. W tym celu wykorzystuję skrót *CTRL* + *SHIFT* + *F*.



Obraz 2. Intellij Gradle Projects

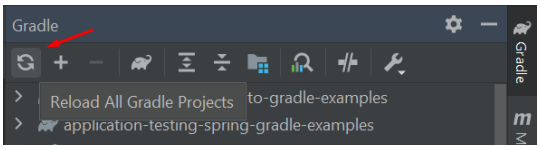
Gdy już zamieniłem wszędzie Gradle Wrapper na wersję 8, upewniłem się jak mam to ustawione w Intellij. W tym celu przeszedłem do ustawień (*CTRL* + *ALT* + *S*) i wyszukałem ustawień Gradle:



Obraz 3. Intellij Gradle Settings

Dla każdego projektu pokazanego tutaj na liście mam ustawione, że *Use Gradle from: 'gradle-wrapper.properties' file*. Zauważ, że ustawiona Java to 17. Przypomnę, że do omówienia zmian w kolejnych wersjach Java przejdziemy w dedykowanym warsztacie.

Teraz przechodzę do zakładki Gradle widocznej po prawej stronie ekranu i wciskam przycisk *Reload All Gradle Projects*:



Obraz 4. Intellij Reload All Gradle Projects

Komputer mi się teraz zaciął tak, że myślałem, że się nie odetnie, ale całe szczęście po ~~tygodniu~~ chwili się udało. Na ekranie nie został wydrukowany żaden błąd.

Co dalej? To samo możemy zrobić, ustawiając Java 19 w ustawieniach Intellij. W moim przypadku projekty nadal działają poprawnie.

W następnej kolejności chcę uruchomić `clean build` dla każdego projektu (czyli rekursywne przeszukiwanie katalogów w celu znalezienia pliku `build.gradle`), który mam zdefiniowany w swoim zajavka repozytorium. W tym celu potrzebuję jakiejś automatyzacji. Chcę zatem wykonać skrypt w Windows CMD, który uruchomi komendę `clean build` i wydrukuje na ekranie rezultat. Do budowania ma zostać wykorzystany Gradle Wrapper (jeżeli taki w projekcie istnieje), jeżeli natomiast nie, ma zostać wykorzystany globalny Gradle.

Przy wykorzystaniu ChatGPT wygenerowałem następujący skrypt:

```
@echo off

echo === PROCESSING START ===

for /r %%d in (.) do (
  pushd "%%d"
  if exist "gradlew.bat" (
    echo === GRADLE WRAPPER FOUND IN: %%d ===
    call gradlew.bat clean build
  ) else if exist "build.gradle" (
    echo === FOUND FILE build.gradle IN: %%d ===
    gradle clean build
  )
  popd
)

echo === PROCESSING END ===
```

Musiałem kilka razy poprawiać ChatGPT albo uściślać wymagania, żeby w końcu osiągnąć efekt, który chcę. Nie udało mi się wygenerować skryptu za pierwszym razem.

Uruchamiam teraz powyższy skrypt w *root* mojego repozytorium, żebym mógł wykonać wszystkie `clean build` i widzę, że wszystko działa poprawnie. To znaczy:

- w późniejszych warsztatach dowiesz się, czym są testy, natomiast testy są uruchamiane przy wykonaniu komendy `clean build`. Testy działały poprawnie z poprzednimi wersjami Gradle i nadal działają poprawnie.
- nie zaobserwowałem żadnego słowa 'ERROR' w informacjach, które są drukowane na ekranie podczas wykonywania skryptu. Są to tak zwane *logi* skryptu. Czym jest logowanie, również dowiesz się w dedykowanym warsztacie.

Po wykonaniu powyższego skryptu można wyszukać na ekranie (lub w pliku, zależy, w jaki sposób uruchomimy komendę) czy pojawiają się takie słowa jak np. *ERROR* albo *FAIL*. W moim przypadku nie pojawiły się żadne błędy, które wynikałyby ze zmiany wersji Gradle, a wszystkie `clean build` zakończyły się sukcesem.

Deprecated Gradle

Przy wykonaniu buildów z tych wszystkich projektów, można natomiast zwrócić uwagę, że na ekranie pojawia się wiadomość:

```
Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.

You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come
from your own scripts or plugins.

See https://docs.gradle.org/8.0.2/userguide/command_line_interface.html#sec:command_line_warnings
```

Co możemy zrobić, gdy wystąpi taka sytuacja? W pierwszej kolejności zapoznać się z tą odpowiedzią.

Żeby zrobić to inaczej niż przez flagę `--warning-mode=all`, możemy obok pliku `gradle.build` dodać plik `gradle.properties` i umieścić w nim taki wpis:

Plik gradle.properties

```
org.gradle.warning.mode=all
```

W moim przypadku na ekranie pojawia się wtedy dodatkowy wpis:

```
> Task :compileJava
The GradleVersion.getNextMajor() method has been deprecated. This is scheduled to be removed in Gradle
9.0. Consult the upgrading guide for further information:
https://docs.gradle.org/8.0.2/userguide/upgrading_version_7.html#org_gradle_util_reports_deprecations
```

Task `:compileJava` pochodzi z pluginu *java*, zatem nie jest to błąd, który powodujemy swoim kodem. Możemy ten błąd ewentualnie wyłączyć ustawieniem:

Plik gradle.properties

```
org.gradle.warning.mode=none
```

A jakieś nowe rzeczy?

Rozczaruję Cię, ale w Gradle 8 nie zostało wprowadzone nic nowego, co jest istotne na tym etapie nauki z naszego punktu widzenia. Dlatego właśnie chciałem poświęcić czas i pokazać Ci, że projekty, które miałem przygotowane w Gradle 7, nadal działają poprawnie.

A jak nie będzie działać?

W przyszłości może wystąpić taka sytuacja, że konfiguracja napisana w Gradle 7 z jakiegoś powodu przestanie działać, gdy zrobimy update np. do Gradle 10. Co wtedy zrobić? Znaleźć taki Migration Guide i poszukać, co uległo zmianie. Na tej podstawie ocenić, co u nas nie działa i co należy poprawić.

Podsumowanie

W programowaniu to jest normalne (można powiedzieć, że jest to chleb codzienny dewelopera), że wersje narzędzi i bibliotek ulegają zmianie. Ciągnie to za sobą takie konsekwencje, że deweloper musi na bieżąco aktualizować wersje narzędzi i bibliotek w swoich projektach i śledzić, że takie zmiany w ogóle występują.

Oczywiście istnieją narzędzia, które umożliwiają taką automatyczną aktualizację wersji i przykładowo, jeżeli będziesz w przyszłości korzystać z GitHub, to dostawca ten posiada rozwiązanie nazwane Dependabot, które służy do automatyzacji aktualizowania wersji zależności.

Spokojnie, jeżeli chodzi o tego GitHuba. Wrócimy do niego w dedykowanym warsztacie, wtedy dowiesz się, co to jest i po co to jest. Chciałem tutaj tylko zaznaczyć, że są narzędzia do automatyzowania podbijania wersji bibliotek i podałem przykład.

Release Notes

Z każdym nowym wdrożeniem jakiegoś narzędzia, twórcy najczęściej przygotowują coś takiego jak Release Notes, czyli informacje o tym, *co uległo zmianie*. W przypadku Gradle 8 możesz takie coś znaleźć tutaj. Z tego miejsca dowiesz się jakie zmiany zostały wprowadzone i jakie błędy zostały naprawione. Błędy znajdziesz na dole w sekcji *Fixed issues*.

To na podstawie takich notatek (Release Notes) oraz Migration Guides, deweloperzy, pracując z dokumentacją, dostosowują swój projekt do nowszych wersji narzędzi, o ile taka konieczność dostosowania w ogóle wystąpi. W naszym przypadku nie wystąpiła, dlatego podałem tylko źródła i wskazówki, na co należy zwracać uwagę.

Filozofia dotycząca kursu

Na koniec tego wpisu chcę zaznaczyć pewną istotną kwestię dotyczącą organizacji materiałów w tym kursie. W kolejnych warsztatach będziemy korzystać w ogromnej większości z Gradle. W notatkach będą się pojawiały przykłady analogiczne napisane w Maven, ale na filmach będziemy korzystać z Gradle, żeby móc stosować krótszą formę zapisu i więcej zmieścić na ekranie ☺. W ramach ćwiczeń zachęcam jednak do pisania tego samego w Gradle i w Maven.

Materiały dotyczące Hibernate, czy Spring (m.in. o tym będziemy rozmawiać w kolejnych warsztatach) zostały przygotowane przy wykorzystaniu Gradle 7. Oznacza to, że w kolejnych warsztatach, za każdym razem, gdy będziemy dodawać kolejne nowe rzeczy, będą one realizowane w Gradle 7. Gradle będzie nam służył jako narzędzie do organizacji projektu i jego sposób działania (oraz sposób myślenia o nim) nie ulegają zmianie z kolejnymi wersjami tego narzędzia.

Z racji, że w tym kursie zależy nam na przedstawieniu Ci sposobu myślenia o konkretnych mechanizmach i narzędziach, będziemy reagować na zmiany wersji narzędzi wtedy, gdy będą one wprowadzały istotne zmiany. Co mam na myśli? Kolejne materiały zostały przygotowane przy wykorzystaniu Gradle 7, ale możesz je również realizować, korzystając z Gradle 8 i wyższych wersji. Jeżeli w kolejnych wersjach będzie pojawiało się coś istotnego, będziemy to dodawać do kursu na odpowiednich etapach.

Czyli jeżeli będziesz oglądać materiały z Gradle 7 i coś Ci nie będzie działać, to spokojnie, na pewnym etapie do tego wrócimy, a Ty albo możesz spróbować rozwiązać problem przy wykorzystaniu wszystkich dostępnych możliwości, albo poczekać do momentu, aż dodamy materiał pokazujący jak radzić sobie z tym problemem.

To samo występuje z resztą w przypadku "czystej" Javy. Bootcamp Zajavka to podstawy programowania w Java, natomiast omówienie funkcjonalności wprowadzanych w kolejnych wersjach Javy następuje dopiero w dedykowanym warsztacie.

Czyli co? Po prostu nam zaufaj i ucz się jak Ci wygodnie. Jak coś nie działa, jak powinno, korzystaj z Internetu, albo rób na takich wersjach, jak pokazujemy ☺.