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.